Department of Computer Science

Technical Report

Ph.D. Dissertation: A Formal Execution Semantics and Rigorous Analytical Approach for Communicating UML Statechart Diagrams

Vitus S.W. Lam

Technical Report 2006-04 **April 2006**

ISSN 1740-9497

Contact Address:
Department of Computer Science
University of Bath
Bath, BA2 7AY
United Kingdom
URL: http://www.cs.bath.ac.uk

ISSN 1740-9497

A Formal Execution Semantics and Rigorous Analytical Approach for Communicating UML Statechart Diagrams

submitted by
Vitus S.W. Lam

for the degree of Doctor of Philosophy
of the
University of Bath

February, 2006

Signature of Author .....
Vitus S.W. Lam

Abstract

The major problem of the official UML semantics is it lacks the precise definition which is required for performing a formal analysis and reasoning on statechart diagrams. This thesis first examines a rigorous approach for the formalization of the execution semantics of communicating UML statechart diagrams in the π-calculus. An integrated approach which is based on the formalized execution semantics is then proposed for the equivalence checking and model checking of statechart diagrams.

Our formalization transforms a subset of UML statechart diagrams as distinct from statecharts into the π-calculus as a number of processes which communicate via a channel-passing interaction paradigm. Checking equivalence of any two statechart diagrams is transformed to a problem of verifying whether the corresponding π-calculus process expressions are equivalent. An equivalence-checking environment which consists of three software tools is described and demonstrated. The environment allows a user to draw statechart diagrams with Poseidon for UML, translate them into π-calculus representations with SC2PiCal and check whether the statechart diagrams are equivalent using the MWB.

Likewise, we put forward a model-checking environment. To verify the correctness of a finite state system represented as multiple interacting statechart diagrams, we specify the design in statechart diagrams using Poseidon for UML, formalize them in the π-calculus, transform the π-calculus expressions into equivalent NuSMV code using PiCal2NuSMV and verify the system automatically using the NuSMV model checker.

Our work illustrates how an integrated approach can provide a more thorough analysis of statechart diagrams. The main contributions of this thesis are:

(i) a formal definition of the execution semantics of statechart diagrams;

(ii) an integration of different formal methods and tools for analyzing statechart diagrams; and

(iii) a practical application of the proposed approach for verifying the correctness of SET/A protocol.

Acknowledgements

I would like to express my gratitude to my supervisor, Julian Padget, for his supervision, support and encouragement throughout my studies at Bath. He introduced me to the field of process algebras, read the manuscripts of my papers and gave helpful suggestions on my dissertation work.

Thanks to all anonymous referees for providing valuable comments and constructive criticisms on the published papers that form the core parts of this thesis.

I am indebted to Angela Cobban for her assistance during my stays in Bath. My thanks also extend to friends who provided their generous support throughout the study period.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Unified Modeling Language (UML) [14, 93, 82, 83] has become the de facto standard for the development of object-oriented systems in industry. UML includes a set of nine diagram types which model different views of a system. UML statechart diagrams [14, 93, 82, 83], which are a variant of Harel's statecharts [42], are a diagrammatic notation for visualizing and expressing the dynamic aspects of a system. They depict how an object responds to various events throughout its lifetime.

UML statechart diagrams are characterized by their graphical syntax and execution semantics. The graphical syntax defines precisely how a statechart diagram is constructed from the notational elements of UML statechart diagrams, whereas the execution semantics specifies the mechanism for processing an event and firing a transition.

Though UML statechart diagrams have a well-defined graphical syntax, their execution semantics is only described in [82, 83] in informal English. A precise definition of the execution semantics is a prerequisite for performing a formal analysis on UML statechart diagrams.

There have been a number of studies on the execution semantics of UML statechart diagrams. Latella et al. [61, 60, 38, 39] have shown how UML statechart diagrams are formalized in extended hierarchical automata (EHA) by unfolding the hierarchical structure of a composite state. Lilius and Paltor [62] have presented a formalization of UML statechart diagrams in PROMELA [48]. Reggio et al. [98] have explored how UML statechart diagrams are translated into Common Algebraic Specification Language (CASL). In addition, Reggio et al. have pointed out that the formalization

of UML statechart diagrams is important as it reveals ambiguities, inconsistencies and incompleteness of the OMG Unified Modeling Language Specification [82].

While much research has been devoted to the formalization of UML statechart diagrams, little research has been done on the integration of different formal methods and software tools for the analysis of UML statechart diagrams. To provide a thorough analysis of UML statechart diagrams, an alternative approach which formalizes UML statechart diagrams in the π-calculus [77, 74] is presented in this thesis. The π-calculus is a channel-passing process algebra which was developed by Milner, Parrow and Walker [77, 74]. It is a formalism for specifying concurrent systems. We adopt the π-calculus for the formalization of the execution semantics of UML statechart diagrams as (i) it provides a compositional approach for modelling the hierarchical structure of a statechart diagram as parallel composition; and (ii) it allows the encoding of the firing priority scheme of statechart diagrams as channel passing between concurrent processes. Our formalization focuses on the major graphical constructs of UML statechart diagrams which consist of various types of transitions and states. These include interlevel transitions, conflicting transitions, non-composite states (basic states), non-concurrent composite states and concurrent composite states.

Although our work shares some similarities with previous studies [61, 60, 38, 39, 62, 98], there are also significant differences. In contrast to the formalization of [61, 60, 38, 39], our approach preserves the hierarchical structure of UML statechart diagrams through the use of parallel composition. The preservation of hierarchical structure reduces the complexity of formalization and retains the levels of abstraction. [62] formalizes the run-to-completion step [82, 83] separately as an algorithm. On the contrary, ours is represented directly in the π-calculus and only a single formalism is needed in our formalization. Unlike the work of [98] which focuses on the use of formalization for identifying ambiguities, inconsistencies and incompleteness of the UML documentation, we emphasize the formal analysis of UML statechart diagrams.

The contribution of our work is not only that a systematic approach for the definition of execution semantics is advocated. More importantly, it throws light on how equivalence checking and model checking [30] of UML statechart diagrams can be carried out. Practical applications of our work include the determination of substitutability of UML statechart diagrams, detection of inconsistency between UML statechart diagrams and verification of the correctness of a system against its specifications.

Proving two statechart diagrams are equivalent is important as it allows us to

distinguish between statechart diagrams and determine when one statechart diagram can substitute for another one. Equivalence checking is a complex process as the number of transitions grows exponentially as the number of states increases.

Similarly, the integration of UML statechart diagrams with model checking is useful as it provides a rigorous approach for verifying the correctness of a model in the design stage. Unlike testing, which is used for finding errors in an implementation, our approach enables the verification of a model against its specifications before the model is implemented.

To automate the equivalence checking and model checking of UML statechart diagrams, an integrated environment based on the formalized execution semantics has been implemented. The integrated environment comprises two translators which provide linkages between UML statechart diagrams and the π-calculus as well as the π-calculus and the NuSMV model checker.

1.1 Achievements

Prior to the submission of thesis, some parts of the thesis have been published as a number of papers:

1. Lam, V.S.W. and Padget, J. Formalization of UML Statechart Diagrams in the π-Calculus. In *Proceedings of 2001 Australian Software Engineering Conference*, pages 213–223. IEEE Computer Society, 2001.
2. Lam, V.S.W. and Padget, J. On Execution Semantics of UML Statechart Diagrams Using the π-Calculus. In *Proceedings of the International Conference on Software Engineering Research and Practice*, pages 877–882. CSREA Press, 2003.
3. Lam, V.S.W. and Padget, J. Analyzing Equivalences of UML Statechart Diagrams by Structural Congruence and Open Bisimulations. In *Proceedings of 2003 IEEE Symposium on Human Centric Computing Languages and Environments*, pages 137–144. IEEE Computer Society, 2003.
4. Lam, V.S.W. and Padget, J. Symbolic Model Checking of UML Statechart Diagrams with an Integrated Approach. In *Proceedings of Eleventh IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 337–346. IEEE Computer Society, 2004.
5. Lam, V.S.W. and Padget, J. Formal Specification and Verification of the SET/A Protocol with an Integrated Approach. In *Proceedings of 2004 IEEE International*

Conference on E-Commerce Technology, pages 229–235. IEEE Computer Society, 2004.

6. Lam, V.S.W. and Padget, J. Analyzing Execution Semantics of Statecharts Variants. In *Proceedings of 8th World Multi-Conference on Systemics, Cybernetics and Informatics*, volume 1, pages 474–478. IIIS, 2004.

Accepted paper which is based on the remaining parts of the thesis is given below:

1. Lam, V.S.W. and Padget, J. An Integrated Environment for Communicating UML Statechart Diagrams. To appear in Proceedings of 3rd ACS/IEEE International Conference on Computer Systems and Applications.

Submitted paper which is based on the content of the thesis is listed as follows:

1. Lam, V.S.W. and Padget, J. Automated Equivalence Checking of UML Statechart Diagrams Using the MWB. Submitted for publication.

Other publications which were written during the PhD study period include:

1. Lam, V.S.W. Book review on Communicating and Mobile Systems: the π-Calculus. *ACM Software Engineering Notes*, 25(1): 121, 2000.
2. Lam, V.S.W. and Padget, J. Consistency Checking of Statechart Diagrams of a Class Hierarchy. To appear in Proceedings of 19th European Conference on Object-Oriented Programming.
3. Lam, V.S.W. and Padget, J. Consistency Checking of Statechart Diagrams and Sequence Diagrams Using the π-Calculus. To appear in Proceedings of Integrated Formal Methods 2005.

1.2 Structure of the Thesis

Chapter 2 recalls the syntax and execution semantics of UML statechart diagrams which are based on the official UML documentation. A comparison of UML statechart diagrams with Harel's statecharts is provided. The limitations and weaknesses of the original execution semantics and previous formalizations are examined in great detail.

An overview of the π-calculus is given in Chapter 3. A structured approach for the formalization of communicating UML statechart diagrams is presented. The correctness of the formalization is proved. Examples are given for illustrating how various graphical constructs of statechart diagrams are encoded in the π-calculus.

A practical application of the formalization is shown in Chapter 4. It illustrates how the equivalence of two statechart diagrams is formally verified by translating them

into the π-calculus. The work described in this chapter is the first part of an integrated approach which unifies equivalence checking and model checking for the analysis of statechart diagrams.

A rigorous approach for verifying the design of a system represented as a number of interacting statechart diagrams against its specifications is discussed in Chapter 5. This chapter constitutes the second part of the integrated approach.

The integrated approach is analyzed and evaluated in Chapters 6 and 7 by using a case study approach. Applications of the integrated approach for the analysis of an agent-based payment protocol and a statechart diagram consisting of a concurrent composite state are illustrated in Chapter 6.

Chapter 7 details the design and implementation of an integrated environment which supports the integrated approach. It contains discussion about the development of two translators, SC2PiCal and PiCal2NuSMV, in which they integrate Poseidon for UML, MWB and NuSMV as an integrated environment for the equivalence checking and model checking of statechart diagrams.

Chapter 8 concludes the thesis. It highlights the contributions of the thesis and sketches lines of future research.

Appendix A is a quick reference to the commands for the integrated environment. Appendix B details the implementation of the two translators SC2PiCal and PiCal2NuSMV.

Chapter 2

Background

Statecharts are a graphical notation which was developed by Harel [42] for visualizing and specifying the dynamic behaviour of a system. The statechart formalism extends the conventional state transition diagrams by incorporating the notions of nested states, orthogonality and broadcast communication mechanism. The statechart diagrams which are part of UML are a variant of Harel's statecharts, but UML statechart diagrams differ from Harel's statecharts in both syntax and execution semantics.

In this chapter, we aim at providing a thorough introduction to statechart formalism. We review the syntax and execution semantics of two major statechart variants: UML statechart diagrams and Harel's statecharts. In particular, the differences in syntax and execution semantics between UML statechart diagrams and Harel's statecharts are highlighted. Examples are given to illustrate how a single diagram is interpreted in different ways using the two execution semantics. Technical issues which are due to the imprecise definition of UML execution semantics are explored. A comparison of our work with previous formalizations of UML execution semantics is provided. An alternative formalization approach and two applications of the formalization on the formal analysis and reasoning of UML statechart diagrams are proposed. In addition, justification for our proposed approach is given in detail.

Material from this chapter has been published as [56].

2.1 Syntax

This section summarizes and compares the graphical syntax of Harel's statecharts and UML statechart diagrams. For a more detailed description of the graphical syntax, the

reader is referred to [14, 93, 82, 83, 42, 44, 35, 89, 43]. In particular, [82, 83, 44] provide a more extensive and complete treatment of the subject than the others.

2.1.1 Syntax of Harel's Statecharts and UML Statechart Diagrams

A statechart or statechart diagram comprises two basic entities: state and transition. A **state** is denoted as a rounded rectangle, while a **transition** is denoted as an arrow labelled with three optional parts: event, guard-condition and action.

In statecharts or statechart diagrams there are three types of states: non-composite states, non-concurrent composite states and concurrent composite states. A **non-composite state** is a basic state that does not have any substates. A **non-concurrent composite state** is a state only one of whose substates is active at any point of execution. In a **concurrent composite state** more than one of its substates, which are in different orthogonal regions separated by dashed lines, are active at the same time at any point of execution.

Each statechart or statechart diagram has a **root state** (top state) which contains all other states of the statechart or statechart diagram. The root state is a non-concurrent composite state in which it is not enclosed within any other state. In contrast to other non-concurrent composite states, it does not have any incoming or outgoing transitions.

An **initial pseudostate** is denoted as a small filled circle with an outgoing transition to the default state of a non-concurrent composite state. Each non-concurrent composite state has only one initial pseudostate which represents the start state for the composite state.

A **transition** connects a source state to a target state. A **guard-condition**, which is a Boolean condition, is evaluated whenever the specified event on the transition occurs. An **action** is an atomic computation that is executed when the transition is fired. Typical examples of an action include the invocation of an operation and the sending of an event to an object in which the behaviour is modelled using a statechart diagram. A transition is fired whenever the specified event is present and the guard-condition holds. The action of the transition, if any, is carried out and the target state is entered. Transitions which are in different orthogonal regions of a concurrent composite state may fire simultaneously.

An **interlevel transition** is a transition which crosses the border of a composite

state. It originates from or terminates on the border of a substate of the composite state. Entering or exiting a substate causes the enclosing state i.e. the composite state to be entered or exited as well.

A **fork** expresses a splitting of control into processes which are running in parallel, whereas a **join** expresses a synchronization of control for processes which are running in parallel. Unlike a fork which has one incoming transition and at least two outgoing transitions, a join has at least two incoming transitions and only one outgoing transition. In UML statechart diagrams, a fork pseudostate and a join pseudostate are both denoted as a short thick bar as shown in Figure 2.1, whereas in Harel's statecharts a fork and a join are represented as Figure 2.2.

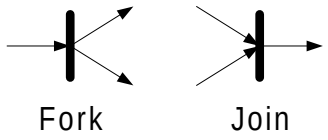

Figure 2.1: Fork and join of UML statechart diagrams

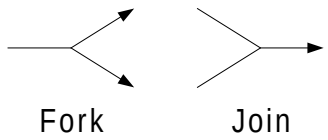

Figure 2.2: Fork and join of Harel's statecharts

Figure 2.3 shows an example of an UML statechart diagram. The root state S_0 is an outermost state which contains all other states of the statechart diagram. The statechart diagram consists of nine non-composite (basic) states $S_1, S_3, V_1, V_2, W_1, W_3$, W_4, T_1 and T_2. The transition t_1 is fired when the event E_1 occurs and the guard-condition $Cond_1$ holds. The source state S_1 is exited, the action $Action_1$ is executed and multiple target states S_2, V_1 and W_1 are entered.

The concurrent composite state S_2 consists of two orthogonal regions (concurrent substates) S_{21} and S_{22} in which the active substates are V_1 and W_1, respectively. Upon the occurrence of event E_3, the transition t_3 is fired, the action $Action_3$ is executed and the non-concurrent composite state W_2 and its default state T_1 are entered. Unlike the

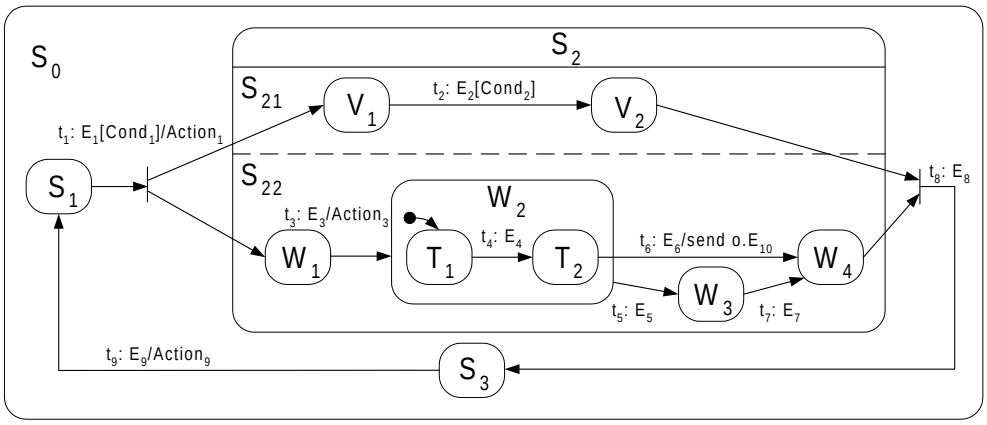

Figure 2.3: Example of an UML statechart diagram

state S_2, the state W_2 has only one active substate T_1. If both V_2 and W_4 are active, the two flows of control in the concurrent substates S_{21} and S_{22} merge into one flow of control upon receipt of the event E_8.

On receipt of event E_5, the transition t_5 is fired. Both the composite state W_2 and its active substate T_1 or T_2 are exited and target state W_3 is entered. The transition t_6 from the source state T_2 to the target state W_4 is an interlevel transition in which the composite state W_2 is also exited whenever the event E_6 occurs. The *send* action then sends an event E_{10} to an object o. Unlike the transition t_5 which is fired when either substates T_1 or T_2 is active, the transition t_6 is fired only when the substate T_2 is active.

As Figure 2.4 shows, there is no difference with Figure 2.3, except the notational distinctions between fork and join and the absence of *send* action in transition t_6. Unlike UML statechart diagrams, Harel's statecharts do not support *send* actions. Instead of sending an event to a particular object, the action of transition t_6 in Figure 2.4 broadcasts the event E_{10}. Each (Harel) statechart is then required to process the event E_{10} even though the event is targeted at another statechart.

2.1.2 Syntactical Differences

Besides the syntactical differences in forks, joins and *send* actions, statechart diagrams extend Harel's statecharts with parameterized events, internal transitions and deferred events (Table 2.1). In statechart diagrams an event is not only a signal, but it may

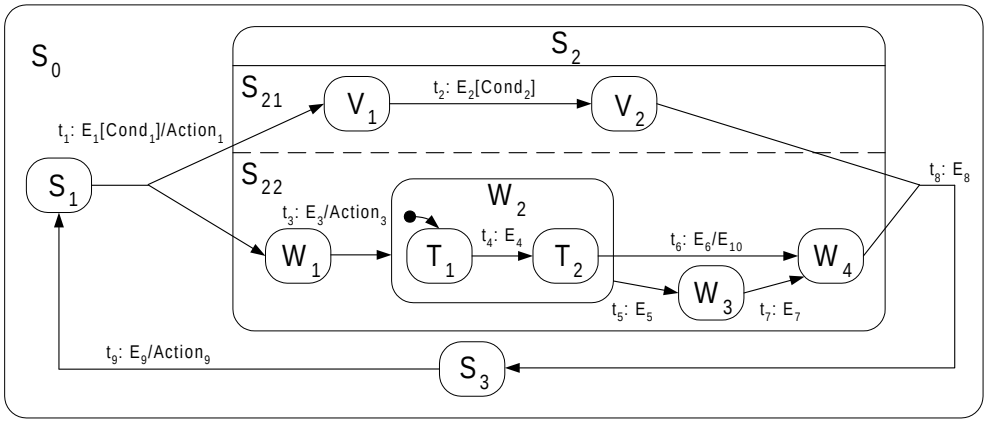

Figure 2.4: Example of a statechart

have arguments in which they are used in the corresponding guard-condition, action and entry action. An internal transition is a transition which executes an action in response to an event without exiting the currently active state. Each state in a statechart diagram may associate with a list of deferred events where responses to them are to be postponed. The deferred events, unlike non-deferred events, do not trigger any transitions. The retention terminates as soon as the statechart diagram enters a state which no longer defers these events.

On the one hand, UML statechart diagrams enrich Harel's statecharts by introducing a number of new features. On the other hand, they also simplify the syntax of Harel's statecharts by removing features, namely (i) the trigger part of a transition of UML statechart diagrams contains only a single event rather than conjunctions of events; and (ii) negated events which model non-occurrence of events are not supported in statechart diagrams.

2.2 Execution Semantics

In this section, we recall the execution semantics of Harel's statecharts and UML statechart diagrams. The adopted execution semantics of UML statechart diagrams is based on the informal UML semantics given in [82, 83].

	UML Statechart Diagrams	Harel's Statecharts
Forks/joins	Yes (with a short thick bar)	Yes (without a short thick bar)
Send actions	Yes	No
Parameterized events	Yes	No
Internal transitions	Yes	No
Deferred events	Yes	No
Conjunctions of events	No	Yes
Negated events	No	Yes

Table 2.1: Comparison of syntax

2.2.1 Execution Semantics of Harel's Statecharts

Harel's statecharts, that are based on STATEMATE semantics [44], adopt a two-level step semantics. A statechart responds to some external events offered by the environment by computing a maximal set of non-conflicting transitions which can be taken in a micro step. The newly generated events, which cannot be sensed in the current micro step, become the input to the next micro step. This chain reaction continues until all internally generated events are processed and the sequence of micro steps forms a macro step. The system moves from one configuration, which is a stable state, to another configuration in a macro step.

The interaction of a statechart with its environment assumes a discrete time model. Depending on the number of steps which are executed in one unit of time, the time model is further classified into synchronous or asynchronous. In the synchronous time model, a statechart interacts with the environment and executes one micro step in one unit of time. On the other hand, in the asynchronous time model a statechart communicates with the environment and executes several micro steps (a macro step) in one unit of time.

In the synchronous time model the execution of a macro step requires non-zero time, whereas in the asynchronous time model the execution of a macro step requires

zero time. No matter whether the micro steps of a macro step are executed at the same instant or not, the duration of an event is limited to one micro step. It does not exist in subsequent micro steps.

2.2.2 Execution Semantics of UML Statechart Diagrams

Each UML statechart diagram is regarded as an abstract machine which has three major components: an event queue, an event dispatcher and an event processor. Events which are offered by the environment are added to the end of the event queue. The event dispatcher chooses, dequeues and provides one event at a time to the event processor. Each event is then executed as a run-to-completion step which ensures that the next event is not dispatched until the processing of the current event is completed. A dispatched event which does not match the trigger events of any transitions in the statechart diagram is discarded i.e. implicitly consumed.

Two transitions are in conflict if they exit the same source state or the source state of one transition is directly or transitively contained in the source state of the other transition. Conflicting transitions are resolved using a lower-first firing priority scheme. A transition originating from a lower-level source state has priority over a conflicting transition in which the source state is higher in the state hierarchy.

A state configuration of a statechart diagram is the maximal set of states which are active at the same time. In a non-concurrent composite state only one of its direct substate is active at the same time, whereas in a concurrent composite state all of its direct substates (orthogonal regions) are active at the same time. When a run-to-completion step is performed, the state configuration changes from one to another.

2.3 Execution Semantics Differences

Besides the syntactical differences, there are also substantial differences in the execution semantics between Harel's statecharts (HSCs) and UML statechart diagrams (UMLSCDs). The differences in the execution semantics are enumerated as follows:

1. UMLSCDs adopt a one-level step semantics (Table 2.2). Each event is executed as a run-to-completion step. Newly generated events in a run-to-completion step are added to the corresponding event queues of the target objects. In contrast, HSCs adopt a two-level step semantics. Newly generated events of a micro step become

	UML Statechart Diagrams (UMLSCDs)	Harel's Statecharts (HSCs)
One/two level step semantics	one level, run-to-completion step	two levels, macro and micro steps
Perfect synchrony hypothesis	No	Yes (asynchronous model) No (synchronous model)
Local consistency	N/A	Yes
Global consistency	N/A	No
Causality	Yes	Yes
Transition refinement	N/A	Yes (asynchronous model) No (synchronous model)
Firing priority scheme	Yes (inner-first)	Yes (outer-first)
Firing priority determination mechanism	relative position of the source state	scope of the transition
No. of events processed at a time	One	One or more
Distinguishing internal and external events	No	Yes
Zero-time semantics	Yes/No	Yes (asynchronous model) No (synchronous model)
Execution of semicolon-separated actions	in sequential order	in parallel
Duration of events	zero or more run-to-completion steps	one micro step

Table 2.2: Comparison of execution semantics

the input to the next micro step. A macro step, which consists of a sequence of micro steps, terminates when all internally generated events are processed.

2. The perfect synchrony hypothesis proposed in [10] assumes that an input event and an output event of a reaction occur at the same time. The atomicity of the reaction is guaranteed. The reaction respects the zero-time semantics and time advances only in between the reactions. In practice, this means that a system is working much faster than its environment. It always processes the generated output event before the occurrence of the next input event. Any changes in a micro step are sensed in the same macro step. In the asynchronous model of STATEMATE, the perfect synchrony hypothesis is assumed. In contrast, changes which are caused by a transition can only be sensed in the next run-to-completion step in UMLSCDs.
3. A contradiction between the trigger part and action part of a transition occurs if the trigger part contains a negated event and the action part contains the same event in a non-negated form. In local consistency [112] the negated event in the trigger part depends only on events generated in previous micro step of the current macro step, but in global consistency [88] it depends on events generated in both previous and future micro steps of the current macro step. Local consistency allows the transition to fire, while global consistency prohibits this. As negated events are not supported in UMLSCDs (see Section 2.1.2), both local and global consistencies are not considered in UML semantics (Table 2.2). In contrast, STATEMATE has adopted local consistency (Table 2.2) and it respects causality [112]. Causality distinguishes a cause (the trigger part) of a transition from its effect (the action part). It ensures that a cause does not depend on events generated by its effect.
4. The substitution of a single transition by a sequence of transitions is known as transition refinement [112]. The asynchronous model of STATEMATE takes zero time to execute both the transition and its refinement, whereas the synchronous model takes non-zero time to execute the refinement. Since there is a difference between the execution times of the transition and its refinement, the synchronous model does not respect transition refinement. Unlike HSCs, UMLSCDs are not based on micro and macro steps. The question of transition refinement does not exist.
5. Both UMLSCDs and HSCs resolve the conflict between two enabled transitions

by a firing priority scheme [112]. In UMLSCDs an inner-first priority scheme is adopted, whereas in HSCs an outer-first priority scheme is adopted. Unlike the priority of a transition in UMLSCDs which depends on the relative position of the source state in the state hierarchy, the priority in HSCs is based on the scope of the transition. The scope of a transition [112, 44] is the lowest common non-concurrent composite state of the source and target of the transition. The priority of HSCs is determined by both the source and target states.

6. In UMLSCDs one event is dispatched and processed at a time. On the contrary, in HSCs a set of events generated in a micro step is available as input to the next micro step in which they are processed at a time.
7. Due to the perfect synchrony hypothesis, the asynchronous model of STATEMATE has to distinguish an internal event from an external event. All internally events generated in micro steps must be processed before an external event is sensed. In the semantics of UMLSCDs, there is no need to distinguish between internal and external events.
8. The asynchronous model of STATEMATE takes zero time to execute a macro step. Time advances in states rather than in transitions. In contrast, UMLSCDs allow a non-zero time semantics and the execution of an action may take time.
9. Both HSCs and UMLSCDs allow the action part of a transition to define a sequence of semicolon-separated actions. In STATEMATE the actions of a transition are executed in parallel, whereas in UMLSCDs the actions of a transition are executed in sequential order. According to [44], the result of updating a common variable by a sequence of actions which are running in parallel is non-deterministic.
10. In UMLSCDs, an event exists in an event queue as long as it is not dispatched by an event dispatcher. The duration of an event depends on the length of the queue and lasts for zero or more run-to-completion steps. In HSCs, an event is only available to the next micro step. The duration of an event is limited to one micro step.

2.4 Different Interpretations of a Single Diagram

This section exemplifies how a single diagram, which conforms to the graphical syntax of both UML statechart diagrams and Harel's statecharts, is interpreted differently

using the execution semantics of UML statechart diagrams and Harel's statecharts. An example is given in Figure 2.5.

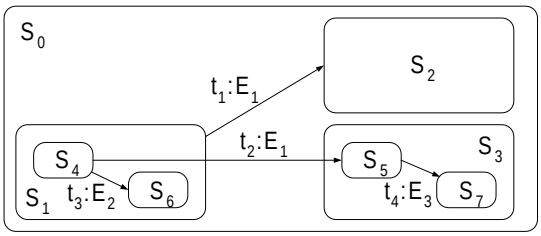

Figure 2.5: Example 1 of different interpretations

Consider the two transitions with labels t_1 and t_2, both depending on E_1. The source states for t_1 and t_2 are S_1 and S_4. State S_4 is lower in the state hierarchy and t_2 is fired according to the execution semantics of UML statechart diagrams. In contrast, transitions t_1 and t_2 have the same scope S_0 according to the STATEMATE semantics and non-determinism arises.

Figure 2.6 provides another example. A sequence of actions, which generates events E_2 and E_3, is executed when t_1 is fired. UML semantics states that E_2 and E_3 are generated in sequential order and only t_2 in T_0 is fired, whereas STATEMATE states that E_2 and E_3 are generated simultaneously and either t_2 or t_3 is fired.

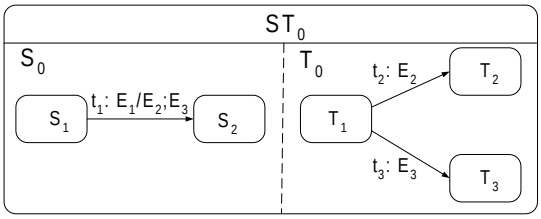

Figure 2.6: Example 2 of different interpretations

Example 3(b) is derived from 3(a) by adding a composite state T_1 and drawing a transition t_6 from T_1 to T_2. However, neither UML semantics nor STATEMATE semantics considers 3(a) and 3(b) as equal. Transition t_5 is fired when event E_1 occurs by adopting the UML semantics, while transition t_6 is fired by adopting the STATEMATE semantics. The non-deterministic behaviour of S_1 is preserved only if the execution semantics does not adopt any firing priority scheme such as the one proposed in [45].

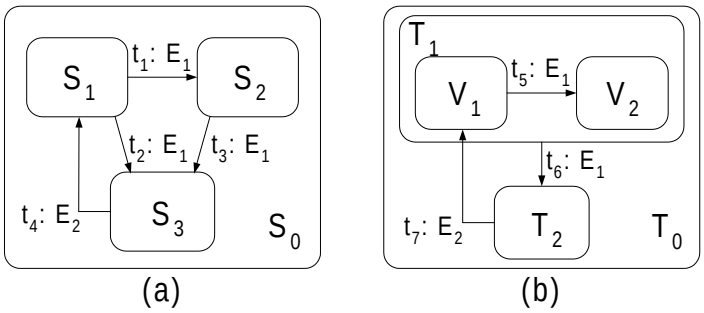

Figure 2.7: Example 3 of different interpretations

Likewise, in Example 4 confusion arises when both events E_1 and E_2 occur at the same time. In a run-to-completion step, only one event is dispatched and processed at a time. Depending on whether event E_1 or E_2 is processed first, either state S_4 or S_5 is entered.

Unlike UML semantics, in STATEMATE semantics both events E_1 and E_2 are available as a set for processing at the same step. Either transition t_1 is fired and state S_2 is entered or transition t_2 is fired and state S_3 is entered.

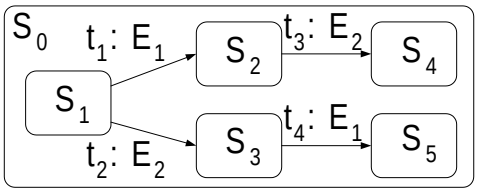

Figure 2.8: Example 4 of different interpretations

2.5 Motivation

After reviewing the execution semantics of UML statechart diagrams and comparing it to the execution semantics of Harel's statecharts, we explain the motivation for studying the formalization of the execution semantics of UML statechart diagrams in this section.

2.5.1 The Importance of a Precise Execution Semantics

The use of informal English for specifying the execution semantics poses a number of problems such as imprecision and incompleteness to the UML documentation. An example of imprecision and incompleteness in the UML documentation is it does not specify precisely and explicitly how statechart diagrams communicate between one another. The original execution semantics focuses on a single statechart diagram instead of multiple communicating statechart diagrams.

To overcome this limitation, we propose the use of a communication mechanism which is by means of a *send* action [93] for communicating between statechart diagrams. Figure 2.9 shows a *send* action in which an event E_2 is sent to an object obj_1 (unicast communication) upon receiving the event E_1. The sending of an event E_2 to a set of objects (multicast communication) is illustrated in Figure 2.10. Similarly, Figure 2.11 shows a *send* action in which an event E_2 is broadcast (broadcast communication) to all objects.

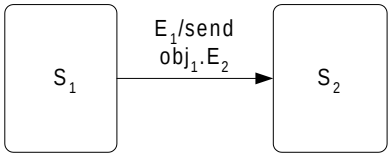

Figure 2.9: Unicast

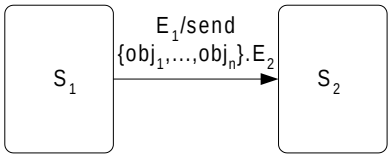

Figure 2.10: Multicast

In our extended execution semantics, we assume that the event queue of each statechart diagram has a unique identifier or address, the transmission of events is reliable and no event is lost. Statechart diagrams interact by sending events to the event queues of other statechart diagrams. The sending of an event E_2 to an object obj_1 is regarded as sending the event E_2 to the event queue of object obj_1. Likewise, the broadcast of

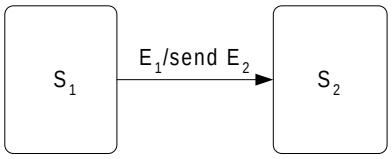

Figure 2.11: Broadcast

an event E_2 to all objects is regarded as sending the event E_2 to the event queues of all objects.

2.5.2 Formal Analysis and Reasoning

Though UML statechart diagrams provide a rich set of notational elements for expressing the behaviour of a system in response to events, the correctness of a system can only be verified using various informal techniques. The lack of a precise execution semantics not only makes it difficult to perform a formal analysis and reasoning on the design of a system, but also hinders the progress of verification tools development.

A well-defined execution semantics is a prerequisite for the integration of UML statechart diagrams with various formal methods and software tools. Through the development of an integrated approach, the equivalences of UML statechart diagrams can be formally proved by checking whether they exhibit the same behaviour in response to various events and the correctness of a system can be formally verified with respect to its specifications. In addition, the formal execution semantics also provides a theoretical foundation for the construction of software tools which support and automate the formal analysis and reasoning.

2.5.3 Review of Previous Formalizations

The major technical issue of the formalization of UML statechart diagrams is to find a suitable formalism for the representation of interlevel transitions. An interlevel transition is a transition which crosses the border of a composite state and it originates from or terminates on the border of a substate of the composite state (Section 2.1). An interlevel transition is like an unstructured goto statement which violates the compositionality semantics.

Latella et al. [61, 60, 38] use extended hierarchical automata (EHA) for the formal-

ization. They unfold interlevel transitions by raising lower-level transitions to higher-level transitions. Similarly, Varró [109] represents statechart diagrams as EHA and defines the operational semantics of EHA using model transition systems.

In our formalization, we introduce a compositional approach for the construction of state hierarchy using the π-calculus. We represent a composite state and its active substates as a number of processes which are running in parallel. An execution of an interlevel transition originating from an active substate is regarded as a self-termination [69] of the π-calculus process representing the active substate. Before the occurrence of the self-termination, the process representing the active substate sends a signal to the process representing the composite state. The composite state is then exited and the target state of the interlevel transition is entered. The π-calculus has an advantage over EHA as it can model state hierarchy as parallel composition and interlevel transition directly, whereas EHA requires an extra table for keeping the true origins of interlevel transitions and other related information such as events, actions, etc.

2.6 Our Approach

A new approach for the formalization of the execution semantics of UML statechart diagrams using the π-calculus is proposed in this thesis. The encoding of UML statechart diagrams in the π-calculus facilitates a formal analysis and reasoning on UML statechart diagrams which includes equivalence checking using Mobility Workbench (MWB) [110, 111] and model checking using NuSMV [23]. In our approach, the design of a system is first documented using a number of communicating statechart diagrams, then translated into π-calculus and finally analyzed using MWB and NuSMV. The reasons for combining these threads as an integrated approach are:

1. The UML has now become a de facto standard for the development of object-oriented systems in industry.
2. As the π-calculus plays an important role in concurrent computing [75] and is a well-established formal notation, we have adopted it as a mathematical notation for defining precisely the informal UML statechart diagrams semantics given in [82, 83].
3. An interlevel transition which is like an unstructured goto statement can be easily represented using the π-calculus without unfolding the hierarchical structure of

UML statechart diagrams.

4. We have adopted the π-calculus instead of Calculus of Communicating Systems (CCS) [73] or value-passing CCS [73] as:
   (i) CCS does not support parameterized agents, parameterized input actions and parameterized output actions. Due to the lack of parameter passing, a specification becomes more complex when compared with value-passing CCS and the π-calculus. A typical 2-buffer example inputting two values based on [34] is represented in CCS as follows:

$$Buffer1 \stackrel{\text{def}}{=} input1.\overline{c1}.Buffer1 + input2.\overline{c2}.Buffer1$$
$$Buffer2 \stackrel{\text{def}}{=} c1.\overline{output1}.Buffer2 + c2.\overline{output2}.Buffer2$$
$$System \stackrel{\text{def}}{=} (Buffer1|Buffer2) \setminus \{c1, c2\}$$

   The corresponding more concise and general π-calculus representation which is capable of inputting n values is given below:

$$Buffer(i, o) \stackrel{\text{def}}{=} i(x).\overline{o}\langle x\rangle.Buffer(i, o)$$
$$System(input, output) \stackrel{\text{def}}{=} (\boldsymbol{\nu} c)(Buffer(input, c)|Buffer(c, output))$$

   (ii) Though value-passing CCS, like the π-calculus, is capable of passing parameters, automated tools which include Edinburgh Concurrency Workbench [107] and Concurrency Workbench of the New Century [31] do not support the analysis of value-passing CCS specifications directly. As a value-passing CCS specification is translated into a CCS specification using a front end tool such as vp [22] before performing the analysis, the size of the specification or the size of the state space actually remains unchanged when compared with the use of CCS specification. The translation also causes the analysis to become more difficult as it requires to relate the CCS specification back to the original value-passing CCS specification.
5. The π-calculus representations provide the flexibility of analyzing statechart diagrams either directly in the π-calculus using MWB or indirectly by translating them into the input language of the NuSMV model checker.
6. Using the π-calculus as an intermediate representation ensures that different implementations of a statechart diagram in various tools are consistent with each other since they are based on the same mathematical model.

7. The π-calculus formalization has advantage over previous formalizations as it provides structural congruence and various types of well-defined behavioural equivalences which form a basis for defining precisely when statechart diagrams are equivalent.
8. The formalization of statechart diagrams is a complex process which is based on their syntax and informally defined execution semantics. The adoption of a 3-tiered architecture using the π-calculus as an intermediate representation reduces the effort by formalizing only once rather than multiple times for different formal methods and software tools. The complexity of the implementation of the intermediate representation in other formal methods and software tools is less as the transformation is mainly syntax-directed.
9. We have chosen the MWB instead of the Open Bisimulation Checker (OBC) Workbench [37, 36] as (i) the MWB supports both monadic and polyadic π-calculi; and (ii) the development of the OBC Workbench has stopped, while the development of the MWB continues.
10. When compared with Symbolic Model Verifier (SMV) [71], NuSMV addresses the state explosion problem in a more efficient way by supporting both binary-decision diagram based (BDD-based [19]) and propositional satisfiability based (SAT-based [11]) model checking.
11. We do not translate a statechart diagram directly into the input language of NuSMV as it is just a programming language rather than a mathematical notation and is not suitable for defining the execution semantics of statechart diagrams.

2.7 Summary and Related Work

In this chapter, we have examined the syntax and execution semantics of UML statechart diagrams and compared them with Harel's statecharts. Examples have been given for illustrating how a single diagram can have very different meanings based on the two execution semantics. The importance of a precise execution semantics definition has been discussed. Issues in previous formalizations and an overview of our approach have been presented.

In [112], the syntax and execution semantics of more than 20 classical statechart variants have been compared. The major limitation of this study is that it is now outdated. Neither the STATEMATE semantics [44] of Harel's statecharts nor UML

semantics of UML statechart diagrams is included in the comparison. In this chapter, we extend [112] by comparing UML statechart diagrams using UML semantics with Harel's statecharts using STATEMATE semantics. Part of our comparison framework is based on [112]. The framework is constructed by (i) extracting comparison criteria that are related to UML statechart diagrams and Harel's statecharts from [112]; and (ii) adding new comparison criteria that are based on concepts used only in UML statechart diagrams and Harel's statecharts. New comparison criteria include parameterized events, internal transitions, deferred events, one/two level step semantics and number of events processed at a time.

Chapter 3

A Formal Execution Semantics for Communicating UML Statechart Diagrams

Numerous research studies have proposed formalisms for modelling concurrent computation. Algebraic formalisms are often referred to as process calculi or process algebras [9]. Depending on whether the interconnection structures of the processes are static or dynamic, process calculi are further classified into non-mobile process calculi and mobile process calculi. Typical examples for non-mobile process calculi are Communicating Sequential Processes (CSP) [47] and Calculus of Communicating Systems (CCS) [73], whereas typical example for mobile process calculi is the π-calculus [77, 74, 76, 87].

In the π-calculus, the concepts of values, variables and channels are integrated as names. The π-calculus extends CCS through the support of name passing. Milner, Parrow and Walker published a landmark paper which describes the monadic π-calculus [77] in 1992. Milner developed the idea a little further by publishing another landmark paper which describes the polyadic π-calculus [74] in 1993. The monadic π-calculus allows the passing of a single name over the channel, while the polyadic π-calculus allows the passing of multiple names over the channel. Sangiorgi [101] has extended previous work by introducing higher-order π-calculus. In higher-order π-calculus, processes may be passed over channels.

This chapter presents a precise execution semantics for communicating UML statechart diagrams. As pointed out in Sections 2.5 and 2.6, we have adopted the π-calculus

as a mathematical notation for execution semantics definition since (i) an interlevel transition which is like an unstructured goto statement can easily be represented in the π-calculus without unfolding the hierarchical structure of UML statechart diagrams as proposed in [61, 60, 38, 39]; and (ii) the π-calculus has structural congruence and various types of well-defined behavioural equivalences which form the basis for defining precisely when statechart diagrams are equivalent.

Unlike previous studies [61, 60, 38, 62], the formalized execution semantics in this chapter is for multiple interacting statechart diagrams rather than a single statechart diagram. It focuses on the communication between statechart diagrams through their event queues.

The remainder of this chapter is organized as follows. The syntax and semantics of the π-calculus are described in Section 3.1. Section 3.2 discusses a formal execution semantics for communicating statechart diagrams which is defined using the π-calculus. The correctness of the formalization is proved in Section 3.3. Examples that illustrate how various graphical constructs of UML statechart diagrams are encoded in the π-calculus are presented in Section 3.4. A comparison with related work is given in Section 3.5. Section 3.7 summarizes the chapter.

The content of Sections 3.1, 3.2, 3.3 and 3.5 is based on [55, 51]. An earlier version of Section 3.4 has been presented in [53].

3.1 The π-Calculus

The π-calculus is a process algebra for specifying concurrent systems in which the processes communicate over channels. As many variants of the π-calculus have been proposed, we briefly review the syntax and semantics of the π-calculus used throughout the thesis. The reader is referred to [76, 87] for details.

We let $\mathcal{A}$ be a set of processes ranged over by P, Q, R, P_i, Q_i, R_i for $i = 1, \ldots, n$, $\mathcal{N}$ be a set of channels (names) ranged over by x, y, x_i, y_i for $i = 1, \ldots, n$ and $\Im$ be a set of process identifiers. A tuple of channels $x_1, x_2, \ldots, x_n$ is abbreviated to $\vec{x}$. The syntax and semantics of π-calculus process expressions are defined as follows:

$x(\vec{y}).P$: is an input prefix which receives channels along channel x and continues as process P with $y_1, y_2, \ldots, y_n$ replaced by the received channels. The input prefix $x().P$ is abbreviated as $x.P$.

$\overline{x}\langle\vec{y}\rangle.P$: is an output prefix which sends channels $y_1, y_2, \ldots, y_n$ along channel x and continues as process P. The output prefix $\overline{x}\langle\rangle.P$ is abbreviated as $\overline{x}.P$.

$P|Q$: represents concurrent processes P and Q are executing in parallel. $\Pi_{i=1}^{n} P_i$ abbreviates $P_1|P_2|\ldots|P_n$.

$P + Q$: represents a non-deterministic choice which either process P or Q proceeds. $\Sigma_{i=1}^{n} P_i$ abbreviates $P_1 + P_2 + \ldots + P_n$.

$(\boldsymbol{\nu}\vec{x})P$: is a restriction which creates new channels $x_1, x_2, \ldots, x_n$ used for communication in process P.

$[x = y]P$: is a matching construct which proceeds as process P if channels x and y are identical; otherwise, behaves like a null process.

$\tau.P$: is an unobservable prefix which performs an internal action τ and continues as process P.

$A(x_1, x_2, \ldots, x_n) \stackrel{\text{def}}{=} P$: denotes a process identifier A which takes n parameters and behaves like process P. Process P may contain occurrences of A.

$\mathbf{0}$: is a null process which cannot perform any actions.

The input prefix $x(\vec{y}).P$ and restriction $(\boldsymbol{\nu}\vec{x})P$ bind $\vec{y}$ and $\vec{x}$ in P, respectively. Unlike the input prefix, the channels $\vec{y}$ in the output prefix $\overline{x}\langle\vec{y}\rangle.P$ are free. The bound names and free names of P are defined as $bn(P)$ and $fn(P)$. The expression $fn(P) \cup fn(Q)$ is abbreviated as $fn(P, Q)$.

Consider for example a client-server model in which a client communicates with a server asynchronously through a reliable transmission medium. Figure 3.1 shows the interactions among the client, medium and server. In the diagram, c, m and s denote the client, merchant and server channels, respectively. The medium first receives a request from the client along channel m and sends the received request to the server along channel s. It then inputs a response from the server along channel m and outputs the received response along channel c. A request req_i gets a response $respn_i$ for $1 \leq i \leq n$.

The behaviour of the client is encoded in the π-calculus as:

$$Client(c, s, m, \overrightarrow{req}) \stackrel{\text{def}}{=}$$
$$\Sigma_{i=1}^{n} \overline{m}\langle req_i, c, s\rangle.c(respn, source).Client(c, s, m, \overrightarrow{req})$$

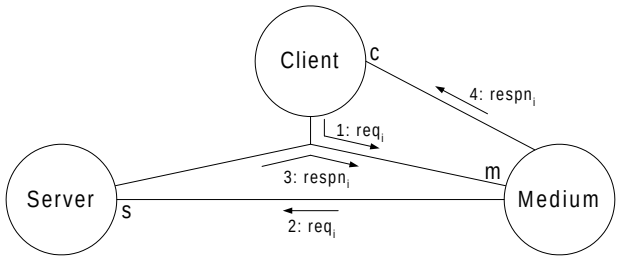

Figure 3.1: Asynchronous client-server model

We model the client as a non-deterministic choice which sends one of the requests req_i for $1 \leq i \leq n$ to the transmission medium along channel m. The client then waits for a response on channel c and behaves as itself.

The transmission medium which resends any received message without message loss is given by:

$$Medium(m) \stackrel{\text{def}}{=} m(msg, source, dest).\overline{dest}\langle msg, source\rangle.Medium(m)$$

The medium receives a message, a source address and a destination address along channel m, sends the received message and source address to the destination address and waits for another message to deliver.

Likewise, we define the behaviour of the server recursively by:

$$Server(s, m, \overrightarrow{req}, \overrightarrow{respn}) \stackrel{\text{def}}{=} s(req, source).(\Sigma_{i=1}^{n}[req = req_i]\overline{m}\langle respn_i, s, source\rangle.Server(s, m, \overrightarrow{req}, \overrightarrow{respn}))$$

The server waits on channel s for a service request, determines what the request is, returns the corresponding response to the client through the transmission medium and continues to serve another service request.

A full specification of the asynchronous communication between the client and the server through a reliable transmission medium is defined by a parallel composition of the client, medium and server as follows:

$$CS\_Model(c, s, m, \overrightarrow{req}, \overrightarrow{respn}) \stackrel{\text{def}}{=} Client(c, s, m, \overrightarrow{req})|Medium(m)|Server(s, m, \overrightarrow{req}, \overrightarrow{respn})$$

3.2 The Formalization

In this section, we examine how a subset of statechart diagrams and an execution semantics of interacting statechart diagrams are formalized in the π-calculus [55, 51]. Notational elements including time events, deferred events and history pseudostates are not considered in our formalization. We do not consider these notational elements as (i) they play a less important role when compared with other notational elements like event, state, guard condition, etc.; and (ii) the adoption of this approach provides a clearer presentation for the formalization.

3.2.1 Formalization of Notational Elements

The strategy of the formalization is to define an appropriate π-calculus representation for each notational element of UML statechart diagrams. Based on the defined mapping, the transformation of notational elements is then repeated until a complete π-calculus specification for a statechart diagram is obtained. As an illustration of the idea, we consider a simple UML statechart diagram as shown in Figure 3.2.

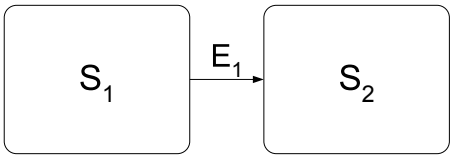

Figure 3.2: Example of a simple UML statechart diagram

We represent the event E_1 as a channel e_1 (Rule 1 of Table 3.1) in the π-calculus. The states S_1 and S_2 are mapped to process identifiers $S_1(step, event_S, \vec{e})$ and $S_2(step, event_S, \vec{e})$ (Rule 2 of Table 3.1) such that $\vec{e}$ abbreviates $e_1, \ldots, e_n$. The π-calculus specification of state S_1 is then defined as:

$$
\begin{aligned}
&S_1(step, event_S, \vec{e}) \stackrel{\text{def}}{=} \\
&\quad event_S(x).([x = e_1]\overline{step}.S_2(step, event_S, \vec{e}) + \\
&\quad \Sigma_{i \neq 1}[x = e_i]\overline{step}.S_1(step, event_S, \vec{e}))
\end{aligned}
$$

The output action $\overline{step}$ models the run-to-completion step defined in the official UML semantics.

Rule	Statechart Diagram	π-Calculus
1	event	channel
2	state	process identifier
3	guard condition	output action
4	action	output action
5	non-concurrent composite state	concurrent processes
6	concurrent composite state	concurrent processes
7	event queue	process
8	dispatcher	process
9	root state	process

Table 3.1: Translation rules

As shown in Table 3.1, we represent guard-condition and action as output action. A guard-condition modelled as an output action accepts either a single channel or a pair of channels. If a single channel is received, the Boolean value is sent on the received channel. If a pair of channels are received, a signal is sent on one of the channels for indicating which Boolean value is stored in the guard-condition. An action represented as an output action corresponds to the invocation of an operation or the sending of an event to an event queue. Likewise, event queue, dispatcher and root state are translated into process. The formalized translation rules are provided in the remainder of this section.

We define $\mathcal{SC}$ as a set of statechart diagrams ranged over by F, G,H,F_i,G_i,H_i for $i = 1, \ldots, n$, $\mathcal{ST}$ as a set of states ranged over by S,T,V,W,S_i,T_i,V_i,W_i for $i = 0, \ldots, n$, $\mathcal{E}$ as a set of events ranged over by $E_1, \ldots, E_n$ and $\mathcal{TR}$ as a set of transitions ranged over by $t_1, \ldots, t_n$. In addition, an infinite set of natural numbers $\mathbb{N}$ and infinite set of positive integers $\mathbb{Z}^+$ are assumed.

The translation of statechart diagrams into the π-calculus is based on the execution semantics discussed in Chapter 2 and the set of rules depicted in Table 3.1. The translation rules in Table 3.1 are formally defined as follows:

Rule 1 *The function $\phi_{event} : \mathcal{E} \rightarrow \mathcal{N}$ maps each event in a statechart diagram to a channel in the π-calculus.*

Let $ST_{Root} \subseteq \mathcal{ST}$ be a set of root states, $ST_{NCS} \subseteq \mathcal{ST}$ be a set of non-composite states,

$ST_{NCCS} \subseteq \mathcal{ST}$ be a set of non-concurrent composite states and $ST_{CCS} \subseteq \mathcal{ST}$ be a set of concurrent composite states.

Rule 2 *The function* $\phi_{state} : \mathcal{ST} \rightarrow \Im$ *returns a unique process identifier for each state. Given a transition which is triggered by an event* $E_1 \in \mathcal{E}$ *connecting* $S_1 \in ST_{NCS}$ *to* $S_2 \in ST_{NCS}$ *then the process identifier* $S_1(step, event_S, \vec{e}) \in \Im$ *representing state* S_1 *is defined as*

$$\begin{aligned} & event_S(x). \\ & ([x = e_1]\overline{step}.S_2(step, event_S, \vec{e}) + \\ & \Sigma_{i \neq 1}[x = e_i]\overline{step}.S_1(step, event_S, \vec{e})) \end{aligned}$$

where $\vec{e}$ *stands for* $e_1, \ldots, e_n$ *and* $\forall a \in \vec{e}.\phi^{-1}_{event}(a) \in \mathcal{E}, \phi_{state}(S_1) = S_1(step, event_S, \vec{e})$, $\phi_{state}(S_2) = S_2(step, event_S, \vec{e})$.

Rule 1 specifies that an event is modelled as a channel in the π-calculus. The inverse of ϕ_{event} denoted by ϕ^{-1}_{event} is a function from $\mathcal{N}$ to $\mathcal{E}$. Rule 2 stipulates that a state is encoded in the π-calculus as a process. The process is regarded as an event processor of the statechart diagram which handles each dispatched event according to the UML step semantics. It determines what the event is by using a number of matching constructs. Both ϕ_{event} and ϕ_{state} are bijective functions.

We define $\mathcal{A}_{in} = \{x(\vec{y}) | x, \vec{y} \in \mathcal{N}\}$ to be a set of input actions and $\mathcal{A}_{out} = \{\overline{x}\langle\vec{y}\rangle | x, \vec{y} \in \mathcal{N}\}$ to be a set of output actions.

Definition 1 *The function* arity: $(\mathcal{A}_{in} \cup \mathcal{A}_{out}) \rightarrow \mathbb{N}$ *returns the number of channels which an input or output action takes as parameters.*

Rule 3 *A mapping between guard-conditions and output actions is defined as* $\phi_{guard} : GCond \rightarrow \{\alpha | \alpha \in \mathcal{A}_{out} \wedge arity(\alpha) \in \{1, 2\}\}$ *where* $GCond$ *is a set of guard-conditions. Given a transition which consists of a guard-condition* g *connecting* $S_1 \in ST_{NCS}$ *to* $S_2 \in ST_{NCS}$ *then the guard-condition is represented as* $\overline{g}\langle x\rangle$ *or* $\overline{g}\langle true\ false\rangle$ *and its Boolean value is tested by either*

$$\begin{aligned} & (\boldsymbol{\nu} x)\overline{g}\langle x\rangle.x(y). \\ & ([y = true]\overline{step}.S_2(step, event_S, \vec{e}, g, true, false) + \\ & [y = false]\overline{step}.S_1(step, event_S, \vec{e}, g, true, false)) \end{aligned}$$

or

$$(\boldsymbol{\nu} true\ false)\overline{g}\langle true\ false\rangle.$$
$$(true.\overline{step}.S_2(step, event_S, \vec{e}, g) +$$
$$false.\overline{step}.S_1(step, event_S, \vec{e}, g))$$

where $\phi^{-1}_{guard}(\overline{g}\langle x\rangle)$, $\phi^{-1}_{guard}(\overline{g}\langle true\ false\rangle) \in GCond$ *and*

$$\phi_{state}(S_1) = \begin{cases} S_1(step, event_S, \vec{e}, g, true, false) & \text{if } \alpha = \overline{g}\langle x\rangle \\ S_1(step, event_S, \vec{e}, g) & \text{if } \alpha = \overline{g}\langle true\ false\rangle \end{cases}$$

$$\phi_{state}(S_2) = \begin{cases} S_2(step, event_S, \vec{e}, g, true, false) & \text{if } \alpha = \overline{g}\langle x\rangle \\ S_2(step, event_S, \vec{e}, g) & \text{if } \alpha = \overline{g}\langle true\ false\rangle \end{cases}$$

$$arity(\alpha) = \begin{cases} 1 & \text{if } \alpha = \overline{g}\langle x\rangle \\ 2 & \text{if } \alpha = \overline{g}\langle true\ false\rangle \end{cases}$$

Rule 4 *Each action representing the invocation of an operation or the sending of a signal to an object is related to an output action in the* π*-calculus by* $\phi_{action} : Act \rightarrow \mathcal{A}_{out}$ *where Act is a set of actions.*

Rules 3 and 4 say that the guard-condition and action of a transition are both represented as an output action and defined as bijective functions. Rule 3 gives two alternatives on how a guard-condition and its evaluation are formalized. One encoding uses two matching constructs to distinguish between the two truth values, whereas the other encoding uses two input actions to determine what the truth value is. The two corresponding alternatives for indicating the Boolean value *true* is stored in a guard-condition g are defined as:

$$True(g, true, false) \stackrel{\text{def}}{=}$$
$$g(x).\overline{x}\langle true\rangle.True(g, true, false)$$

and

$$True(g) \stackrel{\text{def}}{=}$$
$$g(true\ false).\overline{true}.True(g)$$

Similarly, the two corresponding alternatives for denoting the Boolean value $false$ is stored in the guard-condition are obtained by replacing $True$ by $False$, $\overline{x}\langle true\rangle$ by $\overline{x}\langle false\rangle$ and $\overline{true}$ by $\overline{false}$, respectively.

Rule 5 *A non-concurrent composite state S_1 and its active direct substate V_1 are denoted as $\phi_{state}(S_1)|\phi_{state}(V_1)$. Given a transition which is triggered by an event E_1 connecting a non-concurrent composite state S_1 to a non-composite state S_2 where $\phi_{state}(S_1) = S_1(step, event_S, \vec{e}, event_V, pos, neg)$ and $\phi_{state}(V_1) = V_1(event_V, \vec{e}, pos, neg)$, $\phi_{event}(E_1) = e_1$, $\phi_{state}(S_2) = S_2(step, event_S, \vec{e}, pos, neg)$ then the process identifiers $S_1(step, event_S, \vec{e}, event_V, pos, neg)$ and $V_1(event_V, \vec{e}, pos, neg)$ are defined by:*

$$
\begin{aligned}
&S_1(step, event_S, \vec{e}, event_V, pos, neg) \stackrel{\text{def}}{=} \\
&\quad event_S(x).(\boldsymbol{\nu} ack)\overline{event_V}\langle x\ ack\rangle.ack(y). \\
&\quad ([y = pos]\overline{step}.S_2(step, event_S, \vec{e}, pos, neg) + \\
&\quad\ [y = neg]\overline{step}.S_1(step, event_S, \vec{e}, event_V, pos, neg))
\end{aligned}
$$

$$
\begin{aligned}
&V_1(event_V, \vec{e}, pos, neg) \stackrel{\text{def}}{=} \\
&\quad event_V(x\ ack). \\
&\quad ([x = e_1]\overline{ack}\langle pos\rangle + \\
&\quad\ \Sigma_{i\neq 1}[x = e_i]\overline{ack}\langle neg\rangle.V_1(event_V, \vec{e}, pos, neg))
\end{aligned}
$$

Rule 6 *A concurrent composite state S_1 and its active substates V_1, V_2 which are located in 2 different orthogonal regions are denoted as $\phi_{state}(S_1)|\phi_{state}(V_1)|\phi_{state}(V_2)$. Given a transition which is triggered by an event E_1 connecting a concurrent composite state S_1 to a non-composite state S_2* where $\phi_{state}(S_1) = S_1(step, event_S, \vec{e}, event_{V_1}, event_{V_2}, pos, neg, cont_{V_1}, cont_{V_2}, end_{V_1}, end_{V_2})$, $\phi_{state}(V_i) = V_i(event_{V_i}, \vec{e}, pos, neg, cont_{V_i}, end_{V_i})$ for $i = 1, 2$, $\phi_{event}(E_1) = e_1$, $\phi_{state}(S_2) = S_2(step, event_S, \vec{e}, pos, neg)$ then the process identifiers $S_1(step, event_S, \vec{e}, event_{V_1}, event_{V_2}, pos, neg, cont_{V_1}, cont_{V_2}, end_{V_1}, end_{V_2})$ and $V_i(event_{V_i}, \vec{e}, pos, neg, cont_{V_i}, end_{V_i})$ for $i = 1, 2$ are defined by:

$$
\begin{aligned}
&S_1(step, event_S, \vec{e}, event_{V_1}, event_{V_2}, pos, neg, cont_{V_1}, cont_{V_2}, end_{V_1}, end_{V_2}) \stackrel{\text{def}}{=} \\
&\quad event_S(x).(\boldsymbol{\nu} ack_1\ ack_2)\overline{event_{V_1}}\langle x\ ack_1\rangle.\overline{event_{V_2}}\langle x\ ack_2\rangle.ack_1(y_1).ack_2(y_2). \\
&\quad ([y_1 = pos][y_2 = pos]\overline{end_{V_1}}.\overline{end_{V_2}}. \\
&\quad\ \overline{step}.S_2(step, event_S, \vec{e}, pos, neg) + \\
&\quad\ [y_1 = pos][y_2 = neg]\overline{cont_{V_1}}.\overline{cont_{V_2}}. \\
&\quad\ \overline{step}.S_1(step, event_S, \vec{e}, event_{V_1}, event_{V_2}, pos, neg, cont_{V_1}, cont_{V_2}, end_{V_1}, end_{V_2}) +
\end{aligned}
$$

$$
\begin{aligned}
&[y_1 = neg][y_2 = pos]\overline{cont_{V_1}}.\overline{cont_{V_2}}.\\
&\overline{step}.S_1(step, event_S, \vec{e}, event_{V_1}, event_{V_2}, pos, neg, cont_{V_1}, cont_{V_2}, end_{V_1}, end_{V_2}) +\\
&[y_1 = neg][y_2 = neg]\overline{cont_{V_1}}.\overline{cont_{V_2}}.\\
&\overline{step}.S_1(step, event_S, \vec{e}, event_{V_1}, event_{V_2}, pos, neg, cont_{V_1}, cont_{V_2}, end_{V_1}, end_{V_2}))
\end{aligned}
$$

$$
\begin{aligned}
&V_i(event_{V_i}, \vec{e}, pos, neg, cont_{V_i}, end_{V_i}) \stackrel{\text{def}}{=}\\
&\quad event_{V_i}(x\ ack_i).\\
&\quad ([x = e_1]\overline{ack_i}\langle pos\rangle.\\
&\quad (end_{V_i} +\\
&\quad\ cont_{V_i}.V_i(event_{V_i}, \vec{e}, pos, neg, cont_{V_i}, end_{V_i})) +\\
&\quad \Sigma_{i\neq 1}[x = e_i]\overline{ack_i}\langle neg\rangle.cont_{V_i}.\\
&\quad V_i(event_{V_i}, \vec{e}, pos, neg, cont_{V_i}, end_{V_i}))
\end{aligned}
$$

Rules 5 and 6 specify that a composite state and its active substates are denoted as processes which are running in parallel. A non-concurrent composite state is regarded as a special case of a concurrent composite state in which there is only one orthogonal region. The composite state broadcasts any received events to its substates. As the substates process the received event before the composite state, the lowest-first firing priority of UML semantics is preserved in our translation.

Rule 7 *Given a statechart diagram $F \in \mathcal{SC}$, the event queue is represented in the π-calculus as:*

$$
\begin{aligned}
&Queue_0^F(ins, del, empty) \stackrel{\text{def}}{=}\\
&\quad ins(x_1).Queue_1^F(ins, del, empty, x_1) +\\
&\quad empty(t\ f).\overline{t}.Queue_0^F(ins, del, empty)
\end{aligned}
$$

$$
\begin{aligned}
&Queue_n^F(ins, del, empty, x_1, \ldots, x_n) \stackrel{\text{def}}{=}\\
&\quad ins(x_{n+1}).Queue_{n+1}^F(ins, del, empty, x_1, \ldots, x_n, x_{n+1}) +\\
&\quad del(r).\overline{r}\langle x_1\rangle.Queue_{n-1}^F(ins, del, empty, x_2, \ldots, x_n) +\\
&\quad empty(t\ f).\overline{f}.Queue_n^F(ins, del, empty, x_1, \ldots, x_n)
\end{aligned}
$$

where $n \geq 1$.

Rule 8 *For a statechart diagram $F \in \mathcal{SC}$, the encoding of the dispatcher is given by:*

$$
\begin{aligned}
&Dispatch^F(t, f, r, del, empty, execute, complete) \stackrel{\text{def}}{=} \\
&\quad \overline{empty}\langle t\ f\rangle. \\
&\quad (t.Dispatch^F(t, f, r, del, empty, execute, complete) + \\
&\quad\ f.\overline{del}\langle r\rangle.r(event).\overline{execute}\langle event\rangle.complete. \\
&\quad Dispatch^F(t, f, r, del, empty, execute, complete))
\end{aligned}
$$

Rule 9 *Given a statechart diagram $F \in \mathcal{SC}$, we denote its root state S_0 in the π-calculus as:*

$$
\begin{aligned}
&S_0(execute, step, complete, event_S) \stackrel{\text{def}}{=} \\
&\quad execute(x).\overline{event_S}\langle x\rangle.step. \\
&\quad \overline{complete}.S_0(execute, step, complete, event_S)
\end{aligned}
$$

In [82, 83], it is not described how the event queue is implemented. In our formalization, we model the event queue as a FIFO queue (Rule 7) which consists of three channels ins, del and $empty$. The channel $empty$ provides a way for determining whether the queue is empty or not. If the queue is not empty, the dispatcher (Rule 8) deletes an event from the front of the queue, sends it along channel $execute$ and waits for a completion signal from the root state (Rule 9). Unlike an ordinary non-concurrent composite state, a root state is a top-level state that (i) does not have any superstates; (ii) cannot have any outgoing transitions; (iii) always remains active; and (iv) never evolves to any other states.

3.2.2 Formalization of Execution Semantics

We now present the formalization of execution semantics in the π-calculus. A formal treatment of concepts including enabled transition, implicit consumption, conflicting transitions, precedence, firing priority scheme, run-to-completion step, etc. is given in this subsection.

Definition 2 *The function* substates: $\mathcal{ST} \rightarrow 2^{\mathcal{ST}}$ *returns the direct substates that are directly contained in a composite state.*

Definition 3 *The function* States*: $\mathcal{SC} \rightarrow 2^{\mathcal{ST}}$ defined by States(F)=$\{S|S$ is a state of statechart diagram $F\}$ returns the set of states of a statechart diagram $F \in \mathcal{SC}$.*

Definition 4 (State Configuration) *A state configuration of a UML statechart diagram $F \in \mathcal{SC}$ is a set $Config^F \subseteq States(F)$ which satisfies the conditions that:*

(i) $\exists_1 s \in Config^F.s \in ST_{Root}$;

(ii) $\forall s \in Config^F.s \in ST_{NCCS} \Rightarrow (\exists_1 v \in substates(s).v \in Config^F)$; *and*

(iii) $\forall s \in Config^F.s \in ST_{CCS} \Rightarrow (substates(s) \subseteq Config^F)$.

The symbol $\exists_1$ is read as 'there exists exactly one'. Clause (i) states that each state configuration has only one root state. Clause (ii) specifies that exactly one direct substate of a non-concurrent composite state is contained in the state configuration, while clause (iii) means that all direct substates of a concurrent composite state are contained in the state configuration.

Let $A_1(\overrightarrow{x_1}) \rightarrow^* A_2(\overrightarrow{x_2})$ stands for $A_1(\overrightarrow{x_1}) \stackrel{\alpha_1}{\rightarrow} P_1 \stackrel{\alpha_2}{\rightarrow} \ldots \stackrel{\alpha_n}{\rightarrow} P_n \stackrel{\alpha_{n+1}}{\rightarrow} A_2(\overrightarrow{x_2})$ where $A_1(\overrightarrow{x_1}), A_2(\overrightarrow{x_2}) \in \Im, P_1, \ldots, P_n \in \mathcal{A}$ and $\bigwedge_{i=1}^{n+1} \alpha_i \in (\mathcal{A}_{in} \cup \mathcal{A}_{out} \cup \tau)$.

Definition 5 *The function* children: $\Im \rightarrow 2^{\Im}$ *returns all children for a process identifier $A(\vec{x}) \in \Im$ where $\phi^{-1}_{state}(A(\vec{x})) \in \mathcal{ST}$ and $substates(\phi^{-1}_{state}(A(\vec{x}))) = \bigcup_{B(\vec{x}) \in children(A(\vec{x}))} \{\phi^{-1}_{state}(B(\vec{x}))\}$.*

A child of a process identifier corresponds to a direct substate of a composite state. In the above definition $substates(\phi^{-1}_{state}(A(\vec{x})))$ (see Definition 2) returns the direct substates for the state $\phi^{-1}_{state}(A(\vec{x}))$ in which it is also denoted as $\bigcup_{B(\vec{x}) \in children(A(\vec{x}))} \{\phi^{-1}_{state}(B(\vec{x}))\}$.

Definition 6 *The function* descendants, *defined below, returns all process identifiers which have a common ancestor process identifier $A(\vec{x}) \in \Im$.*

$$descendants(A(\vec{x})) = \begin{cases} \emptyset & \text{if } children(A(\vec{x})) \\ & \quad = \emptyset \\ \bigcup_{B(\vec{x}) \in children(A(\vec{x}))} & \text{otherwise} \\ \{B(\vec{x})\} \cup & \\ \quad descendants(B(\vec{x})) & \end{cases}$$

Lemma 1 (Event Processor Configuration) *A configuration of the event processor is a set $Config^{EP} \subseteq \{A(\vec{x})\} \cup descendants(A(\vec{x}))$ where $A(\vec{x}) \in \Im$ and $\phi^{-1}_{state}(A(\vec{x})) \in ST_{Root}$ such that:*

(i) $\exists_1 B(\vec{x}) \in Config^{EP}.\ \phi^{-1}_{state}(B(\vec{x})) \in ST_{Root}$;

(ii) $\forall B(\vec{x}) \in Config^{EP}.\ \phi^{-1}_{state}(B(\vec{x})) \in ST_{NCCS} \Rightarrow (\exists_1 C(\vec{x}) \in children(B(\vec{x})).C(\vec{x}) \in Config^{EP})$; *and*

(iii) $\forall B(\vec{x}) \in Config^{EP}.\ \phi^{-1}_{state}(B(\vec{x})) \in ST_{CCS} \Rightarrow (children(B(\vec{x})) \subseteq Config^{EP})$.

Proof. Follows directly from Rule 2 and Definitions 4,5 and 6.

An event processor configuration and a state configuration correspond to each other by a one-to-one mapping. The following definition specifies when a transition is enabled based on the π-calculus. The process $S_1(step, event_S, \vec{e}, gc_1)$ may proceed unchanged as itself according to the notion of implicit consumption.

Definition 7 (Enabled Transition) *A transition of a statechart diagram $F \in \mathcal{SC}$ consisting of an event $E_1 \in \mathcal{E}$ and a guard-condition $g_1 \in GCond$ connecting $S_1 \in ST_{NCS}$ to $S_2 \in ST_{NCS}$ is defined as:*

$$
\begin{aligned}
&S_1(step, event_S, \vec{e}, gc_1) \stackrel{\text{def}}{=} \\
&\quad event_S(x).([x = e_1](\boldsymbol{\nu} t\ f)\overline{gc_1}\langle t\ f\rangle. \\
&\quad (t.\overline{step}.S_2(step, event_S, \vec{e}, gc_1) + \\
&\quad\ f.\overline{step}.S_1(step, event_S, \vec{e}, gc_1)) + \\
&\quad \Sigma_{i\neq 1}[x = e_i]\overline{step}.S_1(step, event_S, \vec{e}, gc_1))
\end{aligned}
$$

where $\phi_{state}(S_1) = S_1(step, event_S, \vec{e}, gc_1)$ and $\phi_{event}(E_1) = e_1$, $\phi_{guard}(g_1) = \overline{gc_1}\langle t\ f\rangle$, $\phi_{state}(S_2) = S_2(step, event_S, \vec{e}, gc_1)$. The transition is enabled if (i) $S_1(step, event_S, \vec{e}, gc_1) \in Config^{EP}$; (ii) x and e_1 are the same channel such that $[x = e_1]$ is true and other matching constructs $[x = e_i]$ for $i \neq 1$ are false; and (iii) a signal is received along t whenever channels t and f are sent along channel gc_1.

Definition 8 (Implicit Consumption) *An event which causes no transition to enable is implicitly consumed and is implemented by the $\Sigma_{i\neq 1}[x = e_i]\overline{step}.S_1(step, event_S, \vec{e}, gc_1)$ expression (see Definition 7).*

Definition 9 (Conflicting Transitions) *Two enabled transitions t_1,$t_2 \in \mathcal{TR}$ are in conflict, written $t_1 \ddagger t_2$, if one of the following conditions is satisfied:*

(i) They are outgoing transitions of the same source state represented by the same process identifier $A(\vec{x}) \in \Im$.

(ii) They are outgoing transitions of two source states $A(\vec{x}), B(\vec{x}) \in \Im$ such that $(A(\vec{x}) \in descendants(B(\vec{x}))) \vee (B(\vec{x}) \in descendants(A(\vec{x})))$.

Definition 10 (Precedence) *For any two process identifiers $A(\vec{x}), B(\vec{x}) \in \Im$ which represent the source states of two transitions $t_1, t_2 \in \mathcal{TR}$ where $\phi^{-1}_{state}(A(\vec{x}))$, $\phi^{-1}_{state}(B(\vec{x})) \in \mathcal{ST}$, $A(\vec{x})$ precedes $B(\vec{x})$, written $B(\vec{x}) \prec A(\vec{x})$, iff $A(\vec{x}) \in descendants(B(\vec{x}))$. Otherwise, there is no precedence relation between $A(\vec{x})$ and $B(\vec{x})$.*

Definition 11 (Firing Priority Scheme) *The conflict between two enabled transitions is resolved by a firing priority scheme:*

(i) The two enabled transitions which are outgoing transitions of the same source state represented by the same process identifier $A(\vec{x}) \in \Im$ are encoded in the π-calculus as a non-deterministic choice. They have the same firing priority and either one of them is fired.

(ii) For $A(\vec{x}) \in descendants(B(\vec{x}))$ where $A(\vec{x}), B(\vec{x}) \in \Im, B(\vec{x}) \prec A(\vec{x})$ and the transition originating from $A(\vec{x})$ has a higher firing priority than the transition originating from $B(\vec{x})$. $A(\vec{x})$ prohibits the enabled transition which origins from $B(\vec{x})$ from firing by sending a negative acknowledgement along the channel ack (see Rule 5). A similar argument applies to the case $B(\vec{x}) \in descendants(A(\vec{x}))$ where $A(\vec{x}), B(\vec{x}) \in \Im$.

Two enabled transitions are in conflict (Definition 9) if they exit the same source state or one source state is a descendant of another one. The firing priority of a transition depends on the relative position of the source state in the state hierarchy as specified in Definition 10. A lower-level source state has priority over those higher-level ones and an inner-first firing priority scheme (Definition 11) is adopted. In the π-calculus, this is implemented either as a non-deterministic choice or a negative acknowledgement *neg* which is sent by the lower-level source state to the higher-level source state (Rules 5 and 6).

Definition 12 (Run-to-Completion Step) *A run-to-completion step of a UML statechart diagram is defined in the π-calculus as*

$$S_0(execute, step, complete, event_S)|\Pi^n_{i=1}A_i(\overrightarrow{x_i}) \rightarrow^*$$
$$S_0(execute, step, complete, event_S)|\Pi^n_{i=1}A'_i(\overrightarrow{x_i})$$

such that:

(i) $S_0(execute, step, complete, event_S)$ is the root state of the statechart diagram;

(ii) $\bigwedge^n_{i=1}(A_i(\overrightarrow{x_i}) \in \Im) \wedge \bigwedge^n_{i=1}(A'_i(\overrightarrow{x_i}) \in \Im)$;

(iii) $A_i^{'}(\overrightarrow{x_i}) = A_i(\overrightarrow{x_i})$ *when there is no state change;*

(iv) *both* $\{\phi_{state}^{-1}(S_0(execute, step, complete, event_S)), \phi_{state}^{-1}(A_1(\overrightarrow{x_1})), \ldots, \phi_{state}^{-1}(A_n(\overrightarrow{x_n}))\}$ *and* $\{\phi_{state}^{-1}(S_0(execute, step, complete, event_S)), \phi_{state}^{-1}(A_1^{'}(\overrightarrow{x_1})), \ldots, \phi_{state}^{-1}(A_n^{'}(\overrightarrow{x_n}))\}$ *are state configurations;*

(v) *both* $\{S_0(execute, step, complete, event_S)\} \cup \bigcup_{i=1}^{n}\{A_i(\overrightarrow{x_i})\}$ *and* $\{S_0(execute, step, complete, event_S)\} \cup \bigcup_{i=1}^{n}\{A_i^{'}(\overrightarrow{x_i})\}$ *are event processor configurations;*

(vi) *either no two enabled transitions* t_1 *and* t_2 *of* $\Pi_{i=1}^{n} A_i(\overrightarrow{x_i})$ *are in conflict i.e.* $\neg(t_1 \ddagger t_2)$ *or the conflict is resolved by the firing priority scheme;*

(vii) $\neg\exists B(\vec{x}), B^{'}(\vec{x}) \in \Im.(B(\vec{x}) \notin \{S_0(execute, step, complete, event_S)\} \cup \bigcup_{i=1}^{n}\{A_i(\overrightarrow{x_i})\} \wedge B^{'}(\vec{x}) \notin \{S_0(execute, step, complete, event_S)\} \cup \bigcup_{i=1}^{n}\{A_i^{'}(\overrightarrow{x_i})\} \wedge B(\vec{x}) \rightarrow^* B^{'}(\vec{x}) \wedge \{S_0(execute, step, complete, event_S)\} \cup \bigcup_{i=1}^{n}\{A_i(\overrightarrow{x_i})\} \cup \{B(\vec{x})\}$ *and* $\{S_0(execute, step, complete, event_S)\} \cup \bigcup_{i=1}^{n}\{A_i^{'}(\overrightarrow{x_i})\} \cup \{B^{'}(\vec{x})\}$ *are event processor configurations); and*

(viii) $\neg\exists B(\vec{x}), B^{'}(\vec{x}) \in \Im, t_1 \in \mathcal{TR}.$ *(t_1 is an enabled transition of* $B(\vec{x}) \wedge B(\vec{x}) \in \bigcup_{i=1}^{n}\{A_i(\overrightarrow{x_i})\} \wedge B(\vec{x}) \rightarrow^* B^{'}(\vec{x}) \wedge B^{'}(\vec{x}) \notin \bigcup_{i=1}^{n}\{A_i^{'}(\overrightarrow{x_i})\} \wedge (\neg\exists t_2 \in \mathcal{TR}.t_2$ *is an enabled transition* $\wedge\ t_1 \ddagger t_2 \wedge t_2$ *has a higher firing priority)).*

A run-to-completion step in statechart diagrams is defined as a sequence of reductions which transform a set of process identifiers representing the current state configuration into another set of process identifiers representing the next state configuration through the operational semantics of the π-calculus. Indeed, each reduction is inferred from the transition rules of the operational semantics of the π-calculus using a deductive approach based on structural operational semantics (SOS). Conditions (iv) and (v) state that a run-to-completion step reduces a state configuration and an event processor configuration to another state configuration and event processor configuration. Condition (vii) says that a maximal set of states are active at the same time before and after a run-to-completion step. Condition (viii) specifies that an enabled transition is guaranteed to fire and reduce to $B^{'}(\vec{x})$ if it is not in conflict with an enabled transition with a higher firing priority. A maximal set of enabled transitions are fired in a run-to-completion step.

In the remainder of this section, we extend the original execution semantics of statechart diagrams to one which supports multiple interacting statechart diagrams. This makes a step toward developing a sound mechanism for analyzing a distributed system modelled by communicating statechart diagrams.

Definition 13 (System Configuration) *Given a system with a set of $m \in \mathbb{Z}^+$ statechart diagrams $\{F_i | 1 \leq i \leq m\} \subseteq \mathcal{SC}$, the system configuration SConfig is represented as:*

$$\begin{aligned}&\Pi_{i=1}^{m}((\Pi_{j=1}^{n_i} A_j^{F_i}(\vec{x}))|Queue_{k_i}^{F_i}(ins_i, del_i, empty_i, x_1, \ldots, x_{k_i})|\\&S_0^{F_i}(execute_i, step_i, complete_i, event_{S_i})|\\&Dispatch^{F_i}(t_i, f_i, r_i, del_i, empty_i, execute_i, complete_i))\end{aligned}$$

where n_i represents the number of states (processes) in a state configuration (event processor configuration) of F_i and k_i denotes the length of the corresponding event queue for F_i.

Unlike an event processor configuration which determines the state for a single statechart diagram, a system configuration determines the state for a system which consists of multiple statechart diagrams.

For an object obj_1, we denote its associated statechart diagram as $F_1 \in \mathcal{SC}$. The corresponding event queue is denoted in the π-calculus as $Queue_n^{F_1}(ins_1, del_1, empty_1, x_1, \ldots, x_n)$ for $n \geq 0$ according to Rule 7. An event $E_1 \in \mathcal{E}$ is added to the event queue $Queue_n^{F_1}(ins_1, del_1, empty_1, x_1, \ldots, x_n)$ by sending a channel e_1 along the channel ins_1 where $\phi_{event}^{-1}(e_1) = E_1$.

Definition 14 (Send Action) *For a source state $S_1 \in \mathcal{ST}$ and a target state $S_2 \in \mathcal{ST}$ which are connected by a transition $t \in \mathcal{TR}$ consisting of an event trigger $E_1 \in \mathcal{E}$ and a send action $act_1 \in Act$ for sending an event $E_2 \in \mathcal{E}$ to an object obj_1, a set of objects $\{obj_1, \cdots, obj_m\}$ or all n objects in the system are defined as:*

$$S_1(step, event_S, \vec{e}, \overrightarrow{ins}) \stackrel{\text{def}}{=} \begin{cases} event_S(x). & \text{if } act_1 = \\ ([x = e_1]\overline{ins_1}\langle e_2\rangle.\overline{step}.S_2(step, event_S, \vec{e}, \overrightarrow{ins}) + & send\ obj_1.E_2 \\ \Sigma_{i\neq 1}[x = e_i]\overline{step}.S_1(step, event_S, \vec{e}, \overrightarrow{ins})) & \\ event_S(x). & \text{if } act_1 = \\ ([x = e_1]\overline{ins_1}\langle e_2\rangle.\cdots.\overline{ins_m}\langle e_2\rangle. & send\ \{obj_1, \\ \overline{step}.S_2(step, event_S, \vec{e}, \overrightarrow{ins}) + & \cdots, obj_m\}.E_2 \\ \Sigma_{i\neq 1}[x = e_i]\overline{step}.S_1(step, event_S, \vec{e}, \overrightarrow{ins})) & \\ event_S(x). & \text{if } act_1 = \\ ([x = e_1]\overline{ins_1}\langle e_2\rangle.\cdots.\overline{ins_n}\langle e_2\rangle. & send\ E_2 \\ \overline{step}.S_2(step, event_S, \vec{e}, \overrightarrow{ins}) + & \\ \Sigma_{i\neq 1}[x = e_i]\overline{step}.S_1(step, event_S, \vec{e}, \overrightarrow{ins})) & \end{cases}$$

where $m, n \in \mathbb{Z}^+$ *and* $n > m$, $\phi_{state}(S_i) = S_i(step, event_S, \vec{e}, \overrightarrow{ins})$ *for* $i = 1, 2$, $\phi_{event}(E_i) = e_i$ *for* $i = 1, 2$, $\phi_{action}(act_1) = \overline{ins_1}\langle e_2\rangle.\cdots.\overline{ins_k}\langle e_2\rangle$ *for* $k \in \{1, m, n\}$.

The *send* action [93] specifies that the communication between statechart diagrams are classified into (i) a unicast communication in which a statechart diagram sends an event to the event queue of another statechart diagram; (ii) a multicast communication in which a statechart diagram sends an event to the event queues of a group of statechart diagrams; and (iii) a broadcast communication in which a statechart diagram sends an event to the event queues of all statechart diagrams.

Definition 15 (Environment) *The environment of a system* Sys_1 *is an abstract representation of a set of systems which interact with* Sys_1 *through (i) a unicast communication in which an event is sent to the event queue of only one statechart diagram in* Sys_1*; (ii) a multicast communication in which an event is sent to the event queues of a group of statechart diagrams in* Sys_1*; and (iii) a broadcast communication in which an event is sent to the event queues of all statechart diagrams in* Sys_1.

Definition 16 (Transition Relation of a System) *The transition relation* $\Delta_{sys} \subseteq \bigcup_i\{SConfig_i\} \times \bigcup_i\{SConfig_i\}$ *for* $i \in \mathbb{Z}^+$ *of a system* Sys_1 *is a set of ordered pairs representing a change from one system configuration to another system configuration which is due to (i) an interaction between the system and the environment (Definition 15); and (ii) an execution of a run-to-completion step of a statechart diagram in the system (Definition 12).*

Next, we generalize Rules 5 and 6 as a new rule and use it in the next section to reason about the correctness of our formalization. The following rule extends these two preceding rules by considering a composite state of more than two hierarchical levels.

Rule 10 *If* $S \in \mathcal{ST}$ *is a composite state of n hierarchical levels, then each active state at level k-1 and its m active direct substates at level k,* $1 \leq k \leq n$ *and* $m \in \mathbb{Z}^+$*, are represented in the* π*-calculus as:*

$$
\begin{aligned}
&S_{k-1}(step, event_{S_{k-1}}, \vec{e}, event_{S_{k1}}, \ldots, event_{S_{km}}, \widetilde{ch}) \stackrel{\text{def}}{=} \\
&\quad event_{S_{k-1}}(x).(\boldsymbol{\nu}\overrightarrow{ack})\overline{event_{S_{k1}}}\langle x\ ack_1\rangle.\cdots. \\
&\quad \overline{event_{S_{km}}}\langle x\ ack_m\rangle.ack_1(y_1).\cdots.ack_m(y_m). \\
&\quad S^{cont}_{k-1}(step, event_{S_{k-1}}, \vec{e}, event_{s_{k1}}, \ldots, event_{s_{km}}, \widetilde{ch}, \vec{y})
\end{aligned}
$$

$$
\begin{aligned}
&S_{ki}(event_{S_{ki}}, \vec{e}, \widetilde{subch}) \stackrel{\text{def}}{=} \\
&\quad event_{S_{ki}}(x\ ack_i). \\
&\quad ([x = e_1]\overline{ack_i}\langle value\rangle.S^{cont}_{ki}(event_{S_k}, \vec{e}, \widetilde{subch}, value) + \cdots + \\
&\quad\ [x = e_n]\overline{ack_i}\langle value\rangle.S^{cont}_{ki}(event_{S_k}, \vec{e}, \widetilde{subch}, value))
\end{aligned}
$$

where $1 \leq i \leq m$ *and* $value \in \{pos, neg\}$, $\widetilde{ch}$ and $\widetilde{subch}$ *are abbreviations such that*

$$
\widetilde{ch} = \begin{cases} pos, neg & if\ S_{k-1} \in ST_{NCCS} \\ pos, neg, \overrightarrow{cont}, \overrightarrow{end} & if\ S_{k-1} \in ST_{CCS} \end{cases}
$$

$$
\widetilde{subch} = \begin{cases} pos, neg & if\ S_{k-1} \in ST_{NCCS} \\ pos, neg, cont_{ki}, end_{ki} & if\ S_{k-1} \in ST_{CCS} \end{cases}
$$

$$
\begin{aligned}
&S^{cont}_{k-1}(step, event_{S_{k-1}}, \vec{e}, event_{s_{k1}}, \ldots, event_{s_{km}}, \widetilde{ch}, \vec{y}) \stackrel{\text{def}}{=} \\
&\begin{cases}
([y_1 = pos]\overline{step}.S'_{k-1}(step, event_{S_{k-1}}, & if\ S_{k-1} \in ST_{NCCS} \\
\vec{e}, event_{s_{k1}}, \ldots, event_{s_{km}}, \widetilde{ch})+ & \\
[y_1 = neg]\overline{step}.S_{k-1}(step, event_{S_{k-1}}, & \\
\vec{e}, event_{s_{k1}}, \ldots, event_{s_{km}}, \widetilde{ch})) & \\
& \\
([y_1 = pos][y_2 = pos]\ldots[y_m = pos] & if\ S_{k-1} \in ST_{CCS} \\
\overline{end_{k1}}.\overline{end_{k2}}.\cdots.\overline{end_{km}}. & \\
\overline{step}.S'_{k-1}(step, event_{S_{k-1}}, & \\
\vec{e}, event_{s_{k1}}, \ldots, event_{s_{km}}, \widetilde{ch})+ & \\
\Sigma_{i=1}^{2^m-1}[y_1 = V_1][y_2 = V_2]\ldots[y_m = V_m] & \\
\overline{cont_{k1}}.\overline{cont_{k2}}.\cdots.\overline{cont_{km}}. & \\
\overline{step}.S_{k-1}(step, event_{S_{k-1}}, & \\
\vec{e}, event_{s_{k1}}, \ldots, event_{s_{km}}, \widetilde{ch})) &
\end{cases}
\end{aligned}
$$

where $S'_{k-1}(step, event_{S_{k-1}}, \vec{e}, event_{s_{k1}}, \ldots, event_{s_{km}}, \widetilde{ch})$ *represents a target state and* $V_i \in \{pos, neg\}$ *for* $i = 1, \ldots, m$, *no two sequences of matching constructs* $[y_1 = V_1][y_2 = V_2] \ldots [y_m = V_m]$ *are identical,* $\bigcup_{i=1}^{m}\{V_i\} \neq \{pos\}$.

$S_{ki}^{cont}(event_{S_k}, \vec{e}, \widetilde{subch}, value) \stackrel{\text{def}}{=}$

$$\begin{cases} \mathbf{0} & if\ value = pos \wedge S_{k-1} \in ST_{NCCS} \\ S_{ki}(event_{S_{ki}}, \vec{e}, \widetilde{subch}) & if\ value = neg \wedge S_{k-1} \in ST_{NCCS} \\ (end_{ki} + cont_{ki}.S_{ki}(event_{S_{ki}}, \vec{e}, \widetilde{subch})) & if\ value = pos \wedge S_{k-1} \in ST_{CCS} \\ cont_{ki}.S_{ki}(event_{S_{ki}}, \vec{e}, \widetilde{subch}) & if\ value = neg \wedge S_{k-1} \in ST_{CCS} \end{cases}$$

3.3 Correctness of the Formalization

Our formalization is a faithful translation as there is a semantic correspondence between a transition in UML statechart diagrams and reductions in the π-calculus. It covers the essential concepts in the execution semantics of UML statechart diagrams. These include run-to-completion step, firing priority scheme, conflicting transitions, state configuration, and so forth. In addition, our translation also preserves the behavioural properties of statechart diagrams which include a lower-first priority scheme and a maximal set of enabled transitions are fired in a run-to-completion step. Following the approach of [60, 39, 66], we prove the correctness of the formalization as shown below:

Definition 17 *The translation of statechart diagrams into the* π*-calculus is defined by the function* $\phi : \mathcal{SC} \rightarrow 2^{\Im}$ *which represents a group of translation functions as specified by Rules 1–10.*

Theorem 1 *Given a transition* $t_1 \in \mathcal{TR}$ *comprising* $E_1 \in \mathcal{E}, g_1 \in GCond$ *and* $act_1 \in Act$ *connecting* $S_1 \in \mathcal{ST}$ *to* $S_2 \in \mathcal{ST}$ *where* act_1 *is defined as send* $obj_1.E_2$. *The firing of* t_1 *which exits* S_1 *and enters* S_2 *semantically corresponds to a sequence* t_1^{π} *of* n *reductions, written* $t_1 \sim_{tr} t_1^{\pi}$, *in the* π*-calculus where* $n \in \mathbb{Z}^+$.

Proof sketch. Two cases are considered.

Case 1. *$S_1, S_2 \in ST_{NCS}$. The behaviour of S_1 is encoded in the π-calculus as:*

$$
\begin{aligned}
&S_1(step, event_S, \vec{e}, g_1, ins_1) \stackrel{\text{def}}{=} \\
&\quad event_S(x).([x = e_1](\boldsymbol{\nu} t\ f)\overline{g_1}\langle t\ f\rangle. \\
&\quad (t.\overline{ins_1}\langle e_2\rangle.\overline{step}.S_2(step, event_S, \vec{e}, g_1, ins_1) + \\
&\quad\ f.\overline{step}.S_1(step, event_S, \vec{e}, g_1, ins_1)) + \\
&\quad \Sigma_{i\neq 1}[x = e_i]\overline{step}.S_1(step, event_S, \vec{e}, g_1, ins_1))
\end{aligned}
$$

where $\phi_{state}(S_1) = S_1(step, event_S, \vec{e}, g_1, ins_1)$ and $\phi_{event}(E_1) = e_1$, $\phi_{guard}(g_1) = \overline{g_1}\langle t\ f\rangle$, $\phi_{action}(send\ obj_1.E_2) = \overline{ins_1}\langle e_2\rangle$, $\phi_{state}(S_2) = S_2(step, event_S, \vec{e}, g_1, ins_1)$, $\vec{e}$ stands for $e_1, \ldots, e_n$ such that $\forall a \in \vec{e}.\phi^{-1}_{event}(a) \in \mathcal{E}$. Thus, the firing of the transition t_1 represented as $S_1 \stackrel{E_1[g_1]/send\ obj_1.E_2}{\longrightarrow} S_2$ is related to a sequence t_1^π denoted as $S_1(step, event_S, \vec{e}, g_1, ins_1) \stackrel{event_S(x)}{\longrightarrow} \stackrel{[x=e_1]\overline{g_1}\langle t\ f\rangle}{\longrightarrow} \stackrel{t}{\longrightarrow} \stackrel{\overline{ins_1}\langle e_2\rangle}{\longrightarrow} \stackrel{\overline{step}}{\longrightarrow} S_2(step, event_S, \vec{e}, g_1, ins_1)$ and there is a correspondence between t_1 and t_1^π. Conversely, suppose $S_1(step, event_S, \vec{e}, g_1, ins_1) \stackrel{event_S(x)}{\longrightarrow} \stackrel{[x=e_1]\overline{g_1}\langle t\ f\rangle}{\longrightarrow} \stackrel{t}{\longrightarrow} \stackrel{\overline{ins_1}\langle e_2\rangle}{\longrightarrow} \stackrel{\overline{step}}{\longrightarrow} S_2(step, event_S, \vec{e}, g_1, ins_1)$. Then, $S_1 \stackrel{E_1[g_1]/send\ obj_1.E_2}{\longrightarrow} S_2$ and the semantic correspondence holds.

Case 2. *$S_1, S_2 \in ST_{NCS} \cup ST_{NCCS} \cup ST_{CCS} \wedge \neg(S_1 \in ST_{NCS} \wedge S_2 \in ST_{NCS})$. If $S_1 \in ST_{NCCS} \wedge S_2, V_1 \in ST_{NCS} \wedge V_1 \in substates(S_1)$, the firing of the transition corresponds to two sequences of reductions $S_1(step, event_S, \vec{e}, g_1, ins_1, event_V, pos, neg) \stackrel{event_S(x)}{\longrightarrow} \stackrel{\overline{event_V}\langle x\ ack\rangle}{\longrightarrow} \stackrel{ack(y)}{\longrightarrow} \stackrel{[y=pos]\overline{step}}{\longrightarrow} S_2(step, event_S, \vec{e}, g_1, ins_1, pos, neg)$ and $V_1(event_V, \vec{e}, g_1, ins_1, pos, neg) \stackrel{event_V(x\ ack)}{\longrightarrow} \stackrel{[x=e_1]\overline{g_1}\langle t\ f\rangle}{\longrightarrow} \stackrel{t}{\longrightarrow} \stackrel{\overline{ins_1}\langle e_2\rangle}{\longrightarrow} \stackrel{\overline{ack}\langle pos\rangle}{\longrightarrow} \mathbf{0}$. These two sequences of reductions imply $S_1(step, event_S, \vec{e}, g_1, ins_1, event_V, pos, neg) \mid V_1(event_V, \vec{e}, g_1, ins_1, pos, neg) \stackrel{event_S(x)}{\longrightarrow} \stackrel{\tau}{\longrightarrow} \stackrel{[x=e_1]\overline{g_1}\langle t\ f\rangle}{\longrightarrow} \stackrel{t}{\longrightarrow} \stackrel{\overline{ins_1}\langle e_2\rangle}{\longrightarrow} \stackrel{\tau}{\longrightarrow} \stackrel{[ack=pos]\overline{step}}{\longrightarrow} S_2(step, event_S, \vec{e}, g_1, ins_1, pos, neg)$ as $\overline{event_V}\langle x\ ack\rangle$, $event_V(x\ ack), \overline{ack}\langle pos\rangle$ and $ack(y)$ are unobservable actions. Thus, the semantic correspondence holds. Since the converse holds true by an analogous argument and similar arguments hold for other cases where $S_1, S_2 \in ST_{NCS} \cup ST_{NCCS} \cup ST_{CCS} \wedge \neg(S_1 \in ST_{NCS} \wedge S_2 \in ST_{NCS})$, this completes the proof of the statement.*

Theorem 2 *Given $F \in \mathcal{SC}$, there is a semantic correspondence between F and $\phi(F)$ which is written as $F \sim_{sc} \phi(F)$.*

Proof sketch. Let F has n transitions and t_1 be an arbitrary transition of F. Then $t_1 \sim_{tr} t_1^\pi$ such that t_1^π is a corresponding sequence of reductions of t_1 associated with

$\phi(F)$. Since t_1 was an arbitrary transition of F, it follows that $\bigwedge_{i=1}^{n}(t_i \sim_{tr} t_i^{\pi})$ holds. Thus, we can conclude that $F \sim_{sc} \phi(F)$.

Lemma 2 *For any number of hierarchical levels n of a composite state, the lower-first firing priority scheme holds.*
Proof. We consider two cases.

Case 1. *n=0 (A basic state at lowest level). Let m be the number of enabled transitions (Definition 7). Consider the case where m=1. Then the lower-first firing priority scheme (Definition 11) holds as there are no conflicting transitions according to Definition 9. Consider the case where $m \geq 2$ (Definition 11(i)). We proceed by contradiction. Suppose p transitions with the same firing priority for $2 \leq p \leq m$ are fired. But this contradicts the fact that only one process can proceed in a non-deterministic choice of the π-calculus (Section 3.1). Thus, the lower-first firing priority scheme (Definition 11(i)) holds.*

Case 2. *$n \geq 1$ (Definition 11(ii)). The proof is by induction on the number of hierarchical levels n for $n \geq 1$.*

Base case: *n=1. The composite state broadcasts any received events to all its active direct substates and the substates always process the received events before the composite state by returning a positive or negative acknowledgement according to the definition of Rule 10. Thus, the statement is true for n=1.*

Induction step: *Assume that every active direct substate at level n-1 processes the received event before their ancestors. By the definition of Rule 10, a composite state at level n-1 broadcasts any received event to its all active direct substates at level n and the substates process the received event before the composite state. Since all composite states which are located at a level less than level n-1 still have not processed the received event, by induction hypothesis, this completes the induction and the proof of the statement.*

Theorem 3 *For any UML statechart diagram, the lower-first firing priority scheme holds.*
Proof. Since each statechart diagram based on Rule 9 has one root state which is a

composite state and it contains all other states, it follows directly from Lemma 2 that our formalization satisfies the statement.

Lemma 3 *For any number of hierarchical levels n of a composite state, a maximal set of enabled transitions which are not in conflict are fired in a run-to-completion step.*
Proof. In this proof we consider the structure of a composite state as an inverted tree as shown in Figure 3.3.

Case 1. *n=0. Let m be the number of enabled transitions. Consider the case where m=1. Then a maximal set of enabled transitions are fired since there is only one enabled transition and it is fired.*
Consider the case where $m \geq 2$. Suppose there are p enabled transitions for $2 \leq p \leq m$ which are in conflict are fired. But this contradicts the fact that only one process can proceed in a non-deterministic choice of the π-calculus (Section 3.1). Thus, a maximal set of enabled transitions are fired as there are m enabled transitions in conflict and only one of them is fired.

Case 2. *$n \geq 1$. The proof is by induction on the number of hierarchical levels n for $n \geq 1$.*

Base case: *n=1. All active states at level 0 send either a positive or negative acknowledgement to their composite states at level 1 according to Rule 10. Since all transitions of the composite states upon receipt of positive acknowledgements (i.e. not in conflict with enabled transitions at level 0) are fired, a maximal number of enabled transitions are fired at level 1.*

Induction step: *Assume that a maximal number of enabled transitions are fired at all levels less than or equal to n-1. Since one of the enabled transitions of the composite state at level n is fired upon receipt of positive acknowledgements from all active state at level n-1, the maximal number of enabled transitions are fired at level n. As a maximal number of enabled transitions are fired at any levels less than or equal to n-1, by induction hypothesis, this completes the induction and the proof of the statement.*

Theorem 4 *For any UML statechart diagram, a maximal set of enabled transitions*

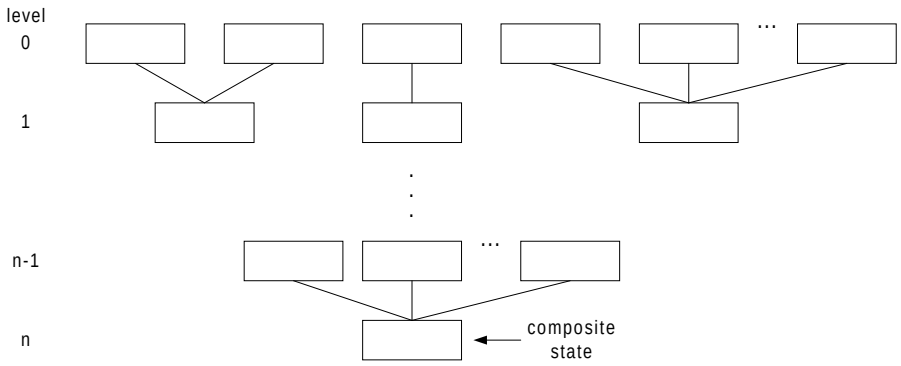

Figure 3.3: Structure of a composite state

which are not in conflict are fired in a run-to-completion step.
Proof. Analogous to Theorem 3.

3.4 Examples

A systematic approach for the translation of UML statechart diagrams into the π-calculus is presented in Section 3.2. This section, following the proposed translation rules, demonstrates how various graphical constructs of statechart diagrams are represented in the π-calculus. The illustrations cover the major graphical constructs including simple transitions, conflicting transitions, entry actions, exit actions, non-concurrent composite states, interlevel transitions and concurrent composite states.

3.4.1 Simple Transitions

A simple transition (Figure 3.4) relates the source state S_1 and target state S_2. It is fired when the event E_1 occurs and the guard-condition $guard_1$ holds. The active state of the statechart diagram is changed from S_1 to S_2 and the action $Action_1$ is executed.

In the π-calculus, the event E_1 is mapped to a channel e_1 (Rule 1). The state S_1 is modelled as a process identifier $S_1(step, event_S, guard_1, action_1, \vec{e})$ which is defined as the following pattern:

$$event_S(x).([x = e_1] \ldots + \Sigma_{i \neq 1}[x = e_i] \ldots)$$

according to Rule 2. The expression $\Sigma_{i \neq 1}[x = e_i]$ signifies the implicit consumption (Definition 8). The guard-condition $guard_1$ is represented as an output action

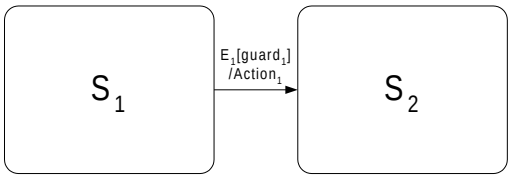

Figure 3.4: A simple transition

$\overline{guard_1}\langle t\ f\rangle$ and tested by a non-deterministic choice:

$$(t.\cdots + f.\cdots)$$

based on Rule 3 to determine its Boolean value. The execution of the action $Action_1$ which models the invocation of an operation is encoded as an output action $\overline{action_1}\langle finish\rangle$ (Rule 4). The completion of the action $Action_1$, which is a signal generated by the action process (operation), is denoted as an input action $finish$. Combining the output action $\overline{action_1}\langle finish\rangle$, input action $finish$ and declaration of channel $finish$, the execution of the action $Action_1$ is expressed as:

$$(\boldsymbol{\nu} finish)\overline{action_1}\langle finish\rangle.finish$$

The end of a run-to-completion step is specified as an output action $\overline{step}$ which interacts with the root state (Rule 9) through the channel *step*. Putting these π-calculus expressions together gives the following π-calculus specification of the simple transition:

$$\begin{aligned}
&S_1(step, event_S, guard_1, action_1, \vec{e}) \stackrel{\text{def}}{=} \\
&\quad event_S(x).([x = e_1](\boldsymbol{\nu} t\ f)\overline{guard_1}\langle t\ f\rangle. \\
&\quad (t.(\boldsymbol{\nu} finish)\overline{action_1}\langle finish\rangle.finish.\overline{step}. \\
&\quad\ S_2(step, event_S, guard_1, action_1, \vec{e}) + \\
&\quad\ f.\overline{step}.S_1(step, event_S, guard_1, action_1, \vec{e})) + \\
&\quad \Sigma_{i\neq 1}[x = e_i]\overline{step}.S_1(step, event_S, guard_1, action_1, \vec{e}))
\end{aligned}$$

The process S_1 takes a parameter list which consists of four channels $step, event_S$, $guard_1$, $action_1$ and a sequence of channels $\vec{e}$. It inputs an event along $event_S$ and

compares the received event with channel e_1. If event E_1 represented by channel e_1 is received, it creates a pair of channels t and f and sends them along channel $guard_1$. It then blocks until a reply is received along the channel t or f. If a signal is received along t, which means that the guard-condition holds and the transition is enabled (Definition 7), it creates a new channel $finish$ and sends the newly created channel $finish$ along channel $action_1$. When a signal is received along the channel $finish$, which indicates that the action is completed, it sends a signal to the root state along channel *step* representing the end of a run-to-completion step and evolves to the process S_2. If a signal is received along f, which means the guard-condition is false, the process S_1 sends a signal along *step* and continues as itself. Similarly, the process S_1 continues unchanged upon receipt of any event other than E_1.

As events, guard-conditions and actions are optional, Figure 3.5 shows some of the other possible patterns for a simple transition. The corresponding π-calculus representations are analogous to the one of Figure 3.4.

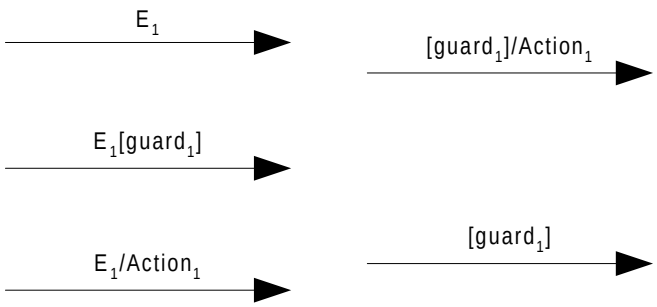

Figure 3.5: Other patterns of a simple transition

3.4.2 Conflicting Transitions

Transitions are in conflict if they exit from the same state (Definition 9(i)). Figure 3.6 shows n deterministic conflicting transitions, written $t_1 \ddagger \ldots \ddagger t_n$ (Definition 9), as the transitions are triggered by different events and guard conditions.

In the π-calculus, deterministic conflicting transitions are modelled as a guarded

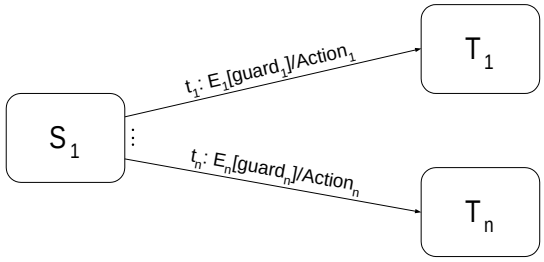

Figure 3.6: Deterministic conflicting transitions

choice (derived from Rule 2). Figure 3.6 is encoded in the π-calculus as:

$$
\begin{aligned}
&S_1(step, event_S, \overrightarrow{guard}, \overrightarrow{action}, \vec{e}) \stackrel{\text{def}}{=} \\
&\quad event_S(x). \\
&\quad (\Sigma_{i=1}^{n}[x = e_i](\boldsymbol{\nu} t_i\ f_i)\overline{guard_i}\langle t_i\ f_i\rangle. \\
&\quad (t_i.(\boldsymbol{\nu} finish)\overline{action_i}\langle finish\rangle.finish.\overline{step}. \\
&\quad T_i(step, event_S, \overrightarrow{guard}, \overrightarrow{action}, \vec{e}) + \\
&\quad f_i.\overline{step}.S_1(step, event_S, \overrightarrow{guard}, \overrightarrow{action}, \vec{e})) + \\
&\quad \Sigma_{i \notin \{1,\ldots,n\}}[x = e_i]\overline{step}.S_1(step, event_S, \overrightarrow{guard}, \overrightarrow{action}, \vec{e}))
\end{aligned}
$$

Depending on which matching construct $[x = e_i]$ for $1 \leq i \leq n$ evaluates true, one of the alternatives is executed.

3.4.3 Entry and Exit Actions

An entry action is executed when a state is entered, whereas an exit action is executed when a state is exited. On receipt of the event E_1 (Figure 3.7), the action $Action_1$ is executed and state S_2 is entered. Similarly, the action $Action_2$ is executed and state S_2 is exited upon receipt of the event E_2.

Alternatively, the behaviour of the entry and exit actions in Figure 3.7 can be modelled by two simple transitions as shown in Figure 3.8.

Since the two statechart diagrams in Figures 3.7 and 3.8 are equivalent, the entry and exit actions in Figure 3.7, like the simple transition in Figure 3.4, are represented

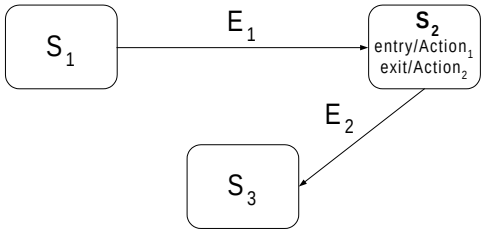

Figure 3.7: Entry and exit actions

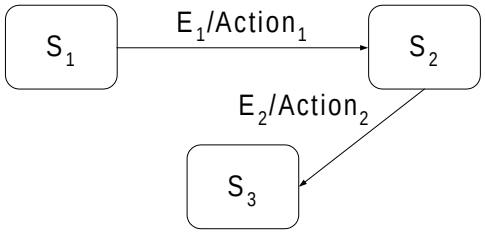

Figure 3.8: Equivalent representations for the entry and exit actions

in the π-calculus as:

$$
\begin{aligned}
&S_1(step, event_S, action_1, action_2, \vec{e}) \stackrel{\text{def}}{=} \\
&\quad event_S(x). \\
&\quad ([x = e_1](\boldsymbol{\nu} finish)\overline{action_1}\langle finish\rangle.finish.\overline{step}. \\
&\quad S_2(step, event_S, action_1, action_2, \vec{e}) + \\
&\quad \Sigma_{i\neq 1}[x = e_i]\overline{step}.S_1(step, event_S, action_1, action_2, \vec{e}))
\end{aligned}
$$

$$
\begin{aligned}
&S_2(step, event_S, action_1, action_2, \vec{e}) \stackrel{\text{def}}{=} \\
&\quad event_S(x). \\
&\quad ([x = e_2](\boldsymbol{\nu} finish)\overline{action_2}\langle finish\rangle.finish.\overline{step}. \\
&\quad S_3(step, event_S, action_1, action_2, \vec{e}) + \\
&\quad \Sigma_{i\neq 2}[x = e_i]\overline{step}.S_2(step, event_S, action_1, action_2, \vec{e}))
\end{aligned}
$$

The execution of an action of a transition as well as an entry or exit action means that multiple actions can be executed in serial. Our formalization can support this as

a sequence of actions in a UML statechart diagram is mapped to a sequence of actions in the π-calculus.

3.4.4 Non-concurrent Composite States

A non-concurrent composite state is a state which contains only one orthogonal region with one or more direct substates. Figure 3.9 shows a non-composite state S_2 which contains n direct substates $V_1, V_2, \ldots, V_n$.

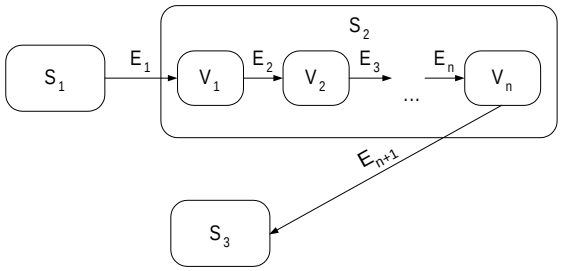

Figure 3.9: Entering and exiting a non-concurrent composite state

The transitions from the state S_1 to the substate V_1 and from the substate V_n to state S_3 are interlevel transitions. They cross the boundary of the non-concurrent composite state S_2. One of them terminates on the substate V_1, while the other one originates from the substate V_n. The π-calculus specification of state S_1 is defined as:

$$
\begin{aligned}
&S_1(step, event_S, \vec{e}, pos, neg) \stackrel{\text{def}}{=} \\
&\quad event_S(x).([x = e_1]\overline{step}. \\
&\quad (\boldsymbol{\nu} event_V)(S_2(step, event_S, \vec{e}, event_V, pos, neg)| \\
&\quad V_1(event_V, \vec{e}, pos, neg)) + \\
&\quad \Sigma_{i \neq 1}[x = e_i]\overline{step}.S_1(step, event_S, \vec{e}, pos, neg))
\end{aligned}
$$

When process S_1 receives e_1, it continues as two concurrent processes S_2 and V_1 as stated in Rule 5.

The non-concurrent composite state S_2 and its active direct substate V_j for $1 \leq j \leq$

n are represented in the π-calculus as follows:

$$
\begin{aligned}
&S_2(step, event_S, \vec{e}, event_V, pos, neg) \stackrel{\text{def}}{=} \\
&\quad event_S(x). \\
&\quad (\boldsymbol{\nu} ack)\overline{event_V}\langle x\ ack\rangle.ack(y). \\
&\quad ([y = pos]\overline{step}.S_3(step, event_S, \vec{e}, pos, neg) + \\
&\quad\ [y = neg]\overline{step}.S_2(step, event_S, \vec{e}, event_V, pos, neg))
\end{aligned}
$$

$$
V_j(event_V, \vec{e}, pos, neg) \stackrel{\text{def}}{=}
\begin{cases}
event_V(x\ ack). & if\ 1 \le j \le \\
([x = e_{j+1}]\overline{ack}\langle neg\rangle.V_{j+1}(event_V, \vec{e}, pos, neg) + & n-1 \\
\ \Sigma_{i \ne j+1}[x = e_i]\overline{ack}\langle neg\rangle.V_j(event_V, \vec{e}, pos, neg)) & \\
\\
event_V(x\ ack). & if\ j = n \\
([x = e_{j+1}]\overline{ack}\langle pos\rangle + & \\
\ \Sigma_{i \ne j+1}[x = e_i]\overline{ack}\langle neg\rangle.V_j(event_V, \vec{e}, pos, neg)) &
\end{cases}
$$

On receiving the channel e_2 (event E_2), the process S_2 passes a tuple which consists of the received channel and an acknowledgement channel ack to the process V_j for $1 \le j \le n$ (active direct substate) along $event_V$. The receipt of a positive acknowledgement pos means that the event does not trigger any transitions at lower-level states and the process S_2 continues as S_3. In contrast, the receipt of a negative acknowledgement neg means that the event triggers a transition at lower-level states and no further transition can be fired by the same event at higher-level states. The process S_2 proceeds as itself and is now ready to receive another event.

The process V_j for $1 \le j \le n-1$ sends out a negative acknowledgement neg and evolves to V_{j+1} upon receiving an event E_{j+1} for $1 \le j \le n-1$. The process V_n responds to the event E_{n+1} by generating a positive acknowledgement pos and terminating itself. The substate V_j for $1 \le j \le n$ precedes the non-concurrent composite state S_2 and is written as $S_2 \prec V_j$ (Definition 10). A lower-first firing priority scheme (Definition 11) is adopted in the π-calculus specifications.

A transition which connects a non-concurrent composite state to a non-composite state (basic state) is semantically equivalent to a number of interlevel transitions which link each of the substates of the non-concurrent composite state to the non-composite

state. A statechart diagram illustrating an exit directly from a non-concurrent composite S_2 is shown in Figure 3.10.

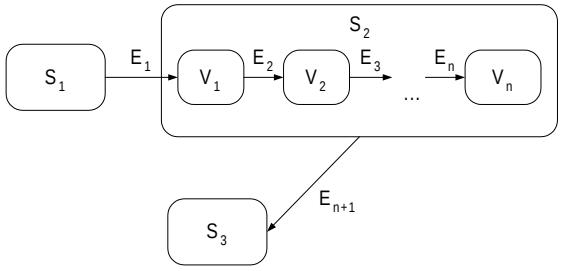

Figure 3.10: Exiting directly from a non-concurrent composite state

The encoding of Figure 3.10 in the π-calculus is similar to the one for Figure 3.9. The main difference is in the representation of the process V_j. As process V_j for $1 \leq j \leq n$ is now capable of responding to the event E_{n+1} by sending out a positive acknowledgement *pos* and terminating itself, the definition of process V_j for $1 \leq j \leq n-1$ is replaced by the following π-calculus specification:

$$V_j(event_V, \vec{e}, pos, neg) \stackrel{\text{def}}{=}$$

$$\begin{cases} event_V(x\ ack). & if\ 1 \leq j \leq \\ ([x = e_{n+1}]\overline{ack}\langle pos\rangle + & n-1 \\ \ [x = e_{j+1}]\overline{ack}\langle neg\rangle.V_{j+1}(event_V, \vec{e}, pos, neg) + \\ \ \Sigma_{i \notin \{n+1, j+1\}}[x = e_i]\overline{ack}\langle neg\rangle.V_j(event_V, \vec{e}, pos, neg)) \\ \\ \ldots \end{cases}$$

3.4.5 Concurrent Composite States

Unlike a non-concurrent composite state, a concurrent composite state consists of multiple orthogonal regions (substates) as illustrated in Figure 3.11. The orthogonal regions, which are separated by dotted lines, contain substates in which only one of them is active at a time.

On receipt of the event E_1, the state S_1 is exited and the states S_2, T_1, T_2, V_1 and W_1 are entered. In contrast, the state S_3 is entered upon receiving the event E_{n+1} if the current active states are S_2, T_1, T_2, V_n and W_n.

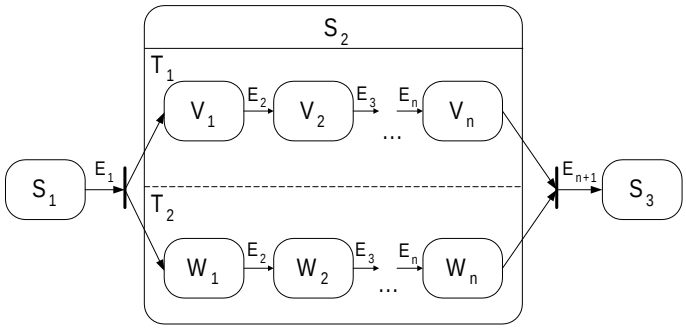

Figure 3.11: Entering and exiting a concurrent composite state

The concurrent composite state S_2 and its active direct substates T_1 and T_2 are encoded in the π-calculus as three concurrent processes. By Rule 5, the non-concurrent composite state T_1 (T_2) and its active direct substate V_1 (W_1) are expressed in the π-calculus as two concurrent processes. The behaviour of state S_1 which evolves to S_2, T_1, T_2, V_1 and W_1 is then specified as:

$$
\begin{aligned}
&S_1(step, event_S, \vec{e}, pos, neg) \stackrel{\text{def}}{=} \\
&\quad event_S(x). \\
&\quad ([x = e_1]\overline{step}. \\
&\quad (\boldsymbol{\nu} event_{T_1}\ event_{T_2}\ event_V\ event_W\ cont_{T_1}\ end_{T_1} \\
&\quad cont_{T_2}\ end_{T_2}\ cont_V\ end_V\ cont_W\ end_W) \\
&\quad (S_2(step, event_S, \vec{e}, event_{T_1}, event_{T_2}, pos, neg, \\
&\quad cont_{T_1}, end_{T_1}, cont_{T_2}, end_{T_2}, cont_V, end_V, cont_W, end_W)| \\
&\quad T_1(event_{T_1}, \vec{e}, event_V, cont_{T_1}, end_{T_1})| \\
&\quad T_2(event_{T_2}, \vec{e}, event_W, cont_{T_2}, end_{T_2})| \\
&\quad V_1(event_V, \vec{e}, pos, neg, cont_V, end_V)| \\
&\quad W_1(event_W, \vec{e}, pos, neg, cont_W, end_W)) + \\
&\quad \Sigma_{i \neq 1}[x = e_i]\overline{step}.S_1(step, event_S, \vec{e}, pos, neg))
\end{aligned}
$$

In contrast to the process S_2 discussed in previous subsection (state S_2 in Figure 3.9), a pair of channels $cont_k$ and end_k for $k \in \{T_1, T_2, V, W\}$ is defined for each process (state) T_1, T_2, V_1, W_1. The process S_2 takes these pairs of channels as parame-

ters and the specification of S_2 is given by:

$$
\begin{aligned}
&S_2(step, event_S, \vec{e}, event_{T_1}, event_{T_2}, pos, neg,\\
&\quad cont_{T_1}, end_{T_1}, cont_{T_2}, end_{T_2}, cont_V, end_V, cont_W, end_W) \stackrel{\text{def}}{=}\\
&\quad event_S(x).\\
&\quad (\boldsymbol{\nu} ack_{T_1}\ ack_{T_2})\overline{event_{T_1}}\langle x\ ack_{T_1}\rangle.\overline{event_{T_2}}\langle x\ ack_{T_2}\rangle.\\
&\quad ack_{T_1}(y).ack_{T_2}(z).\\
&\quad ([y = pos][z = pos]\\
&\quad \overline{end_V}.\overline{end_W}.\overline{end_{T_1}}.\overline{end_{T_2}}.\\
&\quad \overline{step}.S_3(step, event_S, \vec{e}, pos, neg) +\\
&\quad [y = neg][z = neg]\\
&\quad \overline{cont_V}.\overline{cont_W}.\overline{cont_{T_1}}.\overline{cont_{T_2}}.\\
&\quad \overline{step}.S_2(step, event_S, \vec{e}, event_{T_1}, event_{T_2}, pos, neg, cont_{T_1}, end_{T_1},\\
&\quad cont_{T_2}, end_{T_2}, cont_V, end_V, cont_W, end_W) +\\
&\quad [y = pos][z = neg]\\
&\quad \overline{cont_V}.\overline{cont_W}.\overline{cont_{T_1}}.\overline{cont_{T_2}}.\\
&\quad \overline{step}.S_2(step, event_S, \vec{e}, event_{T_1}, event_{T_2}, pos, neg, cont_{T_1}, end_{T_1},\\
&\quad cont_{T_2}, end_{T_2}, cont_V, end_V, cont_W, end_W) +\\
&\quad [y = neg][z = pos]\\
&\quad \overline{cont_V}.\overline{cont_W}.\overline{cont_{T_1}}.\overline{cont_{T_2}}.\\
&\quad \overline{step}.S_2(step, event_S, \vec{e}, event_{T_1}, event_{T_2}, pos, neg, cont_{T_1}, end_{T_1},\\
&\quad cont_{T_2}, end_{T_2}, cont_V, end_V, cont_W, end_W))
\end{aligned}
$$

The process S_2 broadcasts any received event to its direct substates along channels $event_{T_1}$ and $event_{T_2}$. If a positive acknowledgement is received from each of its direct substates, the process S_2 sends a termination signal along channels end_V, end_W, end_{T_1} and end_{T_2} to all its active substates. Otherwise, a continuation signal is sent along channels $cont_V, cont_W, cont_{T_1}$ and $cont_{T_2}$ to all its active substates. Upon receipt of a signal along $cont_k$ for $k \in \{T_1, T_2, V, W\}$, the processes T_1, T_2, V_j and W_j for $1 \leq j \leq n$ continue as themselves. On the contrary, the processes T_1, T_2, V_j and W_j for $1 \leq j \leq n$ terminate themselves on receiving a signal along end_k for $k \in \{T_1, T_2, V, W\}$. The

definitions of processes T_1 and T_2 are given as follows:

$$
\begin{aligned}
&T_1(event_{T_1}, \vec{e}, event_V, cont_{T_1}, end_{T_1}) \stackrel{\text{def}}{=} \\
&\quad event_{T_1}(x\ ack_{T_1}).(\boldsymbol{\nu} ack_V)\overline{event_V}\langle x\ ack_V\rangle.ack_V(y).\overline{ack_{T_1}}\langle y\rangle. \\
&\quad (end_{T_1} + \\
&\quad\ cont_{T_1}.T_1(event_{T_1}, \vec{e}, event_V, cont_{T_1}, end_{T_1}))
\end{aligned}
$$

$$
\begin{aligned}
&T_2(event_{T_2}, \vec{e}, event_W, cont_{T_2}, end_{T_2}) \stackrel{\text{def}}{=} \\
&\quad event_{T_2}(x\ ack_{T_2}).(\boldsymbol{\nu} ack_W)\overline{event_W}\langle x\ ack_W\rangle.ack_W(y).\overline{ack_{T_2}}\langle y\rangle. \\
&\quad (end_{T_2} + \\
&\quad\ cont_{T_2}.T_2(event_{T_2}, \vec{e}, event_W, cont_{T_2}, end_{T_2}))
\end{aligned}
$$

The process T_1 (T_2) sends any received event to its active direct substate along channel $event_V$ ($event_W$), waits for an acknowledgement, outputs it along ack_{T_1} (ack_{T_2}) to the process S_2 and blocks until a reply is received along end_{T_1} (end_{T_2}) or $cont_{T_1}$ ($cont_{T_2}$). The direct substates V_j and W_j for $1 \leq j \leq n$ are specified by:

$V_j(event_V, \vec{e}, pos, neg, cont_V, end_V) \stackrel{\text{def}}{=}$

$$
\begin{cases}
\begin{aligned}
&event_V(x\ ack_V). && if\ 1 \leq j \leq \\
&([x = e_{j+1}]\overline{ack_V}\langle neg\rangle.cont_V. && n-1 \\
&\ V_{j+1}(event_V, \vec{e}, pos, neg, cont_V, end_V) + \\
&\ \Sigma_{i \neq j+1}[x = e_i]\overline{ack_V}\langle neg\rangle.cont_V. \\
&\ V_j(event_V, \vec{e}, pos, neg, cont_V, end_V))
\end{aligned} \\
\\
\begin{aligned}
&event_V(x\ ack_V).([x = e_{j+1}]\overline{ack_V}\langle pos\rangle. && if\ j = n \\
&(end_V + \\
&\ cont_V.V_j(event_V, \vec{e}, pos, neg, cont_V, end_V)) + \\
&\ \Sigma_{i \neq j+1}[x = e_i]\overline{ack_V}\langle neg\rangle.cont_V. \\
&\ V_j(event_V, \vec{e}, pos, neg, cont_V, end_V))
\end{aligned}
\end{cases}
$$

$W_j(event_W, \vec{e}, pos, neg, cont_W, end_W) \stackrel{\text{def}}{=}$

$$\begin{cases} event_W(x\ ack_W). & if\ 1 \leq j \leq \\ ([x = e_{j+1}]\overline{ack_W}\langle neg\rangle.cont_W. & n-1 \\ \ W_{j+1}(event_W, \vec{e}, pos, neg, cont_W, end_W)+ & \\ \ \Sigma_{i\neq j+1}[x = e_i]\overline{ack_W}\langle neg\rangle.cont_W. & \\ \ W_j(event_W, \vec{e}, pos, neg, cont_W, end_W)) & \\ & \\ event_W(x\ ack_W).([x = e_{j+1}]\overline{ack_W}\langle pos\rangle. & if\ j = n \\ (end_W+ & \\ \ cont_W.W_j(event_W, \vec{e}, pos, neg, cont_W, end_W))+ & \\ \ \Sigma_{i\neq j+1}[x = e_i]\overline{ack_W}\langle neg\rangle.cont_W. & \\ \ W_j(event_W, \vec{e}, pos, neg, cont_W, end_W)) & \end{cases}$$

A negative acknowledgement is returned by process V_j (W_j) if an event E_{j+1} for $1 \leq j \leq n-1$ is received. Upon receiving an event E_{n+1}, the process V_n (W_n) sends out a positive acknowledgement *pos* and waits for a decision that either terminates itself or proceeds as itself.

Since processes T_1 and T_2 are only responsible for relaying any received event, an alternative approach as defined by Rule 6 which optimizes the π-calculus representations by eliminating both of them such that any received event is sent directly from S_2 to V_j and W_j for $1 \leq j \leq n$ along $event_V$ and $event_W$ is shown below:

$S_1(step, event_S, \vec{e}, pos, neg) \stackrel{\text{def}}{=}$

$event_S(x).$

$([x = e_1]\overline{step}.$

$(\boldsymbol{\nu} event_V\ event_W\ cont_V\ end_V\ cont_W\ end_W)$

$(S_2(step, event_S, \vec{e}, event_V, event_W, pos, neg, cont_V, end_V, cont_W, end_W)|$

$\ V_1(event_V, \vec{e}, pos, neg, cont_V, end_V)|$

$\ W_1(event_W, \vec{e}, pos, neg, cont_W, end_W)) +$

$\Sigma_{i\neq 1}[x = e_i]\overline{step}.S_1(step, event_S, \vec{e}, pos, neg))$

$$
\begin{aligned}
&S_2(step, event_S, \vec{e}, event_V, event_W, pos, neg, cont_V, end_V, cont_W, end_W) \stackrel{\text{def}}{=} \\
&\quad event_S(x). \\
&\quad (\boldsymbol{\nu} ack_V\ ack_W)\overline{event_V}\langle x\ ack_V\rangle.\overline{event_W}\langle x\ ack_W\rangle. \\
&\quad ack_V(y).ack_W(z). \\
&\quad ([y = pos][z = pos]\overline{end_V}.\overline{end_W}. \\
&\quad \overline{step}.S_3(step, event_S, \vec{e}, pos, neg) + \\
&\quad [y = neg][z = neg]\overline{cont_V}.\overline{cont_W}. \\
&\quad \overline{step}.S_2(step, event_S, \vec{e}, event_V, event_W, pos, neg, cont_V, end_V, cont_W, end_W) + \\
&\quad [y = pos][z = neg]\overline{cont_V}.\overline{cont_W}. \\
&\quad \overline{step}.S_2(step, event_S, \vec{e}, event_V, event_W, pos, neg, cont_V, end_V, cont_W, end_W) + \\
&\quad [y = neg][z = pos]\overline{cont_V}.\overline{cont_W}. \\
&\quad \overline{step}.S_2(step, event_S, \vec{e}, event_V, event_W, pos, neg, cont_V, end_V, cont_W, \\
&\quad end_W))
\end{aligned}
$$

The processes T_1 and T_2 as well as their corresponding channels $event_{T_1}$, $event_{T_2}$, $cont_{T_1}$, $end_{T_1}, cont_{T_2}$ and end_{T_2} are no longer needed in the new definitions of the processes S_1 and S_2. The process S_2 sends a received event along $event_V$ and $event_W$ instead of $event_{T_1}$ and $event_{T_2}$. The representations of V_j and W_j for $1 \leq j \leq n$ remain unchanged.

3.5 Related Work

There have been a number of studies such as [44, 112, 88, 67, 104, 72] on the execution semantics of statecharts [42, 43]. In contrast to a statechart, each UML statechart diagram is associated with an abstract machine which has three major components: an event queue, an event dispatcher and an event processor. The execution semantics of statechart diagrams is based on the run-to-completion step rather than the micro and macro steps of classical statecharts. In statechart diagrams only one event is available at the beginning of each step, whereas in statecharts a set of events are available at the beginning of each step. Due to the somewhat different semantics of statecharts, we limit our comparison to some representative works of UML statechart diagrams which include Latella et al. [61, 60, 38, 39], Lilius and Paltor [62] and Reggio [98]. We do

not consider other works such as von der Beeck [113] which has not specified explicitly what semantics is adopted for the formalization.

Our formalization uses parallel composition for representing the state hierarchy and channel passing for modelling the firing priority scheme. An interlevel transition terminating on a substate of a composite state is encoded as a split into two or more concurrent processes, while an interlevel transition originating from a substate of a composite state is modelled as a self-termination of the process which denotes the substate. Unlike the π-calculus which can directly represent interlevel transitions, Latella et al. unfold interlevel transitions by raising lower-level transitions to higher-level transitions, add labels to the new transitions and define these labels as a table. In contrast to our approach, their encoding of interlevel transitions is more complex.

Lilius and Paltor formalize the run-to-completion step separately as an algorithm, whereas ours is represented directly in the π-calculus and only a single formalism is needed in our formalization.

In [61, 60, 38, 62] the execution semantics is only for a single statechart diagram. In contrast, our proposed execution semantics deals with both a single statechart diagram and multiple interacting statechart diagrams.

The multicharts semantics in [39] is quite different from our proposed execution semantics. Firstly, in our model statechart diagrams and the environment interact in three different types of communication. Secondly, we consider an event queue as one of the components of an abstract machine rather than as part of the environment.

In contrast to the work of Reggio et al. which focuses on the use of the formalization for identifying ambiguities and incompleteness of the UML documentation [82, 83], we emphasize the application of the formalization. In particular, this study is on equivalence checking and model checking.

Other closely related study includes the work of Ray et al. [96]. Ray et al. encode hierarchical state machines (HSMs) as hierarchical process algebra (HPA) which is an extension of a sublanguage of CCS. The sublanguage, unlike CCS, does not include parallel composition, restriction and relabelling. In order to model interlevel transitions, Ray et al. extend the sublanguage by introducing the concepts of embedding, entry points and exit points. An interlevel transition is then regarded as comprising several parts which combine together through the entry and exit points. The extension of this approach to statecharts requires the integration of HPA with another process algebra called Statecharts Process Language (SPL) defined in [66]. In contrast to the approach

of Ray et al. which requires the introduction of new operators specially designed for modelling the interlevel transitions, our encoding is simple, clear and clean as it only makes use of the existing operators of the π-calculus. In addition, the π-calculus, unlike HPA, is a well-established process algebra which facilitates the integration with other formal methods and tools.

3.6 Extensibility of the Approach

Though our formalization is only limited to a subset of UML statechart diagrams, extensions for covering history pseudostates, state references and deferred events are possible. A shallow pseudostate, which keeps track of the most recently active substate of a composite state, is encoded as a process for storing the last active substate. Likewise, a deep history pseudostate is modelled as a number of concurrent processes based on the number of levels of a composite state. By storing a current active state through the use of a process, the modelling of a state reference is also made possible.

The representation of deferred events requires two queues which are identical to the one defined by Rule 7. One of the queues is for keeping a list of deferrable events for a state. The list of deferrable events is updated whenever a state is entered. If an event does not trigger any transitions and is a member of the deferrable events, it is inserted into another queue which stores a list of deferred events. A deferred event is only deleted from the deferred event queue and added back to the event queue as if the event has not been dispatched when the deferred event can fire a transition. As pointed out in [62], the introduction of the deferred event queue prevents a deferred event which does not trigger any transitions to be infinitely dispatched from the event queue.

3.7 Summary

A formal definition of the execution semantics is a prerequisite for developing equivalence-checking and model-checking tools for statechart diagrams In this chapter, we have formalized the execution semantics of statechart diagrams in the π-calculus. An extension to UML semantics for communicating statechart diagrams has been proposed. The research problems identified in Section 2.5, concerning the imprecision and incompleteness of the UML documentation as well as the representation of interlevel transitions,

have been addressed. The formalization presented in this chapter forms a basis for the equivalence checking and model checking of statechart diagrams which are discussed in the next two chapters.

Chapter 4

Equivalences of Statechart Diagrams

In the π-calculus, to determine whether two processes are able to imitate the behaviour of each other, the concepts of structural congruence and bisimulation have been proposed. Depending on when name instantiation of an input prefix is performed, bisimulation can be further classified into early bisimulation, late bisimulation, open bisimulation, etc.

Providing a formal method to prove that two statechart diagrams are equivalent is important in the sense that it allows us to distinguish between statechart diagrams and determine when one statechart diagram can substitute for another one. In this chapter, we study how equivalences of statechart diagrams are formally proved using the π-calculus.

The equivalences of statechart diagrams are classified into three different types: isomorphism, strong behavioural equivalence and weak behavioural equivalence. Isomorphism treats two statechart diagrams as equivalent if they have the same structure, whereas strong and weak behavioural equivalences consider two statechart diagrams as equivalent if they have the same (observable) behaviour. Strong behavioural equivalence equates statechart diagrams with a similar structure and the same behaviour. In contrast, weak behavioural equivalence equates statechart diagrams which have a different structure but exhibit the same behaviour.

The rest of this chapter is structured as follows. Section 4.1 is a review of notation used throughout this chapter. In Sections 4.2–4.4, various types of equivalences

of statechart diagrams based on structural congruence and open bisimulations are introduced. Examples are provided to illustrate how equivalences of statechart diagrams are formally verified. Section 4.5 explores how to specify event channels are distinct using the notion of distinction. A comparison with related work and a summary are given in Sections 4.6 and 4.7.

An earlier version of this chapter first appeared as a conference paper [54] in the proceedings of HCC 2003 and then included in [51].

4.1 Notation

The standard notation for expressing operational semantics in the π-calculus is a labelled transition. Here we review the syntax and semantics of labelled transitions [111, 87] which are used in the definitions of bisimulations. An infinite set of matching constructs (defined in Section 3.1) ranged over by $N_1, \ldots, N_m, \ldots, N_n$ is defined. We use L, M, N to denote finite match sequences which comprise zero or more matching constructs. Conventions related to labelled transitions are now summarized as follows:

$P \xrightarrow{\alpha} P'$: the execution of action α and process P becomes P'.

$P \Longrightarrow P'$: process P becomes P' after zero or more internal actions.

$P \stackrel{\alpha}{\Longrightarrow} P'$: is equivalent to $P \Longrightarrow \xrightarrow{\alpha} \Longrightarrow P'$.

$$P \stackrel{\widehat{\alpha}}{\Longrightarrow} P' : \begin{cases} P \stackrel{\alpha}{\Longrightarrow} P' & if\ \alpha \neq \tau \\ \\ P \Longrightarrow P' & if\ \alpha = \tau \end{cases}$$

$P \stackrel{M,\alpha}{\rightsquigarrow} P'$: if the match sequence M is true, action α is executed and process P becomes P'.

$$P \stackrel{M,\alpha}{\rightsquigarrow\!\!\!\rightsquigarrow} P' : \begin{cases} P \stackrel{N_1,\tau}{\rightsquigarrow} \ldots \stackrel{N_m,\tau}{\rightsquigarrow} \stackrel{L,\alpha}{\rightsquigarrow} \stackrel{N_{m+1},\tau}{\rightsquigarrow} \ldots \stackrel{N_n,\tau}{\rightsquigarrow} P' & if\ \alpha \neq \tau \\ where\ m, n \geq 0\ and \\ value\ of\ M = (\wedge_{i=1}^{n} N_i) \wedge L \\ \\ P \stackrel{N_1,\tau}{\rightsquigarrow} \ldots \stackrel{N_n,\tau}{\rightsquigarrow} P' & if\ \alpha = \tau \\ where\ n \geq 0\ and \\ value\ of\ M = \wedge_{i=1}^{n} N_i \end{cases}$$

4.2 Isomorphism

In the following, we recall the definition of structural congruence which is defined in [77, 74, 76, 87]. A type of equivalence of statechart diagrams named isomorphism is then introduced. An example is given to illustrate how statechart diagrams which are categorized as isomorphic can be formally verified by first translating them into the π-calculus.

Definition 18 *Two processes P and Q are structurally congruent, written $P \equiv Q$, if they satisfy the following axioms:*

(i) P and Q are alpha convertible.

(ii) P and Q can be converted to each other by reordering of terms in a summation.

(iii) P and Q are parallel compositions which can be converted to each other by reordering of terms based on laws of commutativity and associativity.

(iv) P and Q can be converted to each other by scope extrusion law i.e. we regard $R_1|(\boldsymbol{\nu}x)R_2$ equals $(\boldsymbol{\nu}x)(R_1|R_2)$ if x is different from any free name (channel) in R_1.

(v) P and Q can be converted to each other by reordering the sequence of channel declarations i.e. we regard $(\boldsymbol{\nu}x\ y)R$ equals $(\boldsymbol{\nu}y\ x)R$.

Clause (i) states that processes P and Q differ only in the choice of bound names (channels). Clause (iv) extends the scope of channel x to processes R_1 and R_2.

Definition 19 (Isomorphism) *Two statechart diagrams F and G are isomorphic, written $F \cong G$, if and only if there is a bijection $f : States(F) \rightarrow States(G)$ which is independent of the spatial representations of F and G such that each transition of F with source state S_1 and target state S_2 is mapped to a transition of G with source state T_1 and target state T_2 where $S_1, S_2 \in States(F)$ and $T_1, T_2 \in States(G), T_1 = f(S_1), T_2 = f(S_2)$, both transitions are labelled with the same event, guard-condition and action.*

Theorem 5 *Given $F, G \in \mathcal{SC}, F \cong G \leftrightarrow \phi(F) \equiv \phi(G)$.*
Proof sketch.($\rightarrow$) Suppose $F \cong G$. Since $F \cong G, S_1$ corresponds to T_1, S_2 corresponds to T_2, the transition of F which connects source state S_1 and target state S_2 is mapped to the transition of G which connects source state T_1 and target state T_2 where $S_1, S_2 \in States(F)$ and $T_1, T_2 \in States(G)$, both transitions are labelled with the same event,

guard-condition and action. But S_2 and T_2 were arbitrary target states of transitions of S_1 and T_1, respectively. Therefore, the process definitions of $\phi_{state}(S_1)$ and $\phi_{state}(T_1)$ are identical (having same structure) with only differences in the names of the process identifiers and the ordering of terms. Thus, if $F \cong G$ then $\phi(F) \equiv \phi(G)$.
($\leftarrow$) Suppose $\phi(F) \equiv \phi(G)$. Suppose $F \cong G$ is false. Since $\phi(F) \equiv \phi(G)$, $\phi(F)$ can be transformed to $\phi(G)$ by reordering of terms or renaming process identifiers. Then $F \cong G$ as F can be transformed to G through the same transformation by changing the spatial representation or renaming states. But this contradicts the fact that $F \cong G$ is false. Thus, if $\phi(F) \equiv \phi(G)$ then $F \cong G$.

The spatial representations (geometry) of statechart diagrams and choice of state names are unimportant when considering the equivalence of statechart diagrams. Two statechart diagrams are isomorphic if they have the same structure in terms of the π-calculus.

We adopt this approach as the π-calculus, unlike other syntactic representations, has defined precisely when the structures of two π-calculus expressions are the same using structural congruence (Definition 18). Given two statechart diagrams expressed in the π-calculus, we can simply apply the axioms in Definition 18 for determining whether they are isomorphic.

Theorem 6 *The relation $\cong$ is an equivalence.*
Proof. Reflexivity and symmetry are obvious. For transitivity, let F, G and H be arbitrary statechart diagrams. Suppose $F \cong G$ and $G \cong H$. Since $F \cong G$ and $G \cong H$, it follows that $\phi(F) \equiv \phi(G)$ and $\phi(G) \equiv \phi(H)$. As $\equiv$ is transitive, we get $\phi(F) \equiv \phi(H)$ and we can conclude that $F \cong H$. Thus, $\cong$ is transitive.

Figure 4.1 shows two statechart diagrams which are exactly the same with the exception of different spatial representations and change of state names. F_1 and G_1 can be transformed into each other by rotating 180° along the axis i.e. mirroring and renaming of states. To prove that the two statechart diagrams are isomorphic, we first translate them into the π-calculus and then prove that the π-calculus expressions are structurally congruent.

As event queue, event dispatcher and root state are represented in the same way for all statechart diagrams, there is no need to consider them during the equivalence checking process.

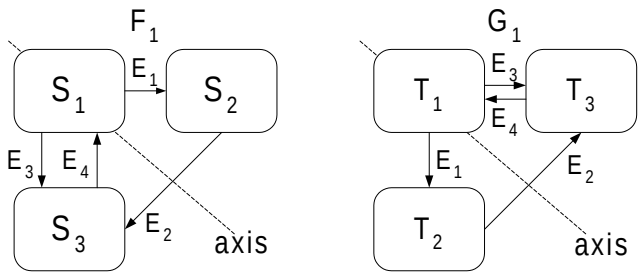

Figure 4.1: Example 1

The three states in statechart diagram F_1 are encoded in the π-calculus as three processes S_1, S_2 and S_3. Similarly, the events in F_1 are denoted in the π-calculus as four channels named e_1, e_2, e_3 and e_4. A signal is sent along channel *step* to the root state defined in Rule 9 for indicating the end of a step. The π-calculus specifications of S_1, S_2 and S_3 are defined as:

$$
\begin{aligned}
&S_1(step, event_S, \vec{e}) \stackrel{\text{def}}{=} \\
&\quad event_S(x).([x = e_1]\overline{step}.S_2(step, event_S, \vec{e}) + \\
&\quad [x = e_3]\overline{step}.S_3(step, event_S, \vec{e}) + \\
&\quad \Sigma_{i \notin \{1,3\}}[x = e_i]\overline{step}.S_1(step, event_S, \vec{e}))
\end{aligned}
$$

$$
\begin{aligned}
&S_2(step, event_S, \vec{e}) \stackrel{\text{def}}{=} \\
&\quad event_S(x).([x = e_2]\overline{step}.S_3(step, event_S, \vec{e}) + \\
&\quad \Sigma_{i \neq 2}[x = e_i]\overline{step}.S_2(step, event_S, \vec{e}))
\end{aligned}
$$

$$
\begin{aligned}
&S_3(step, event_S, \vec{e}) \stackrel{def}{=} \\
&\quad event_S(x).([x = e_4]\overline{step}.S_1(step, event_S, \vec{e}) + \\
&\quad \Sigma_{i \neq 4}[x = e_i]\overline{step}.S_3(step, event_S, \vec{e}))
\end{aligned}
$$

Applying the same translation approach to G_1, the π-calculus expressions of G_1 are the same as F_1 with the exception that the ordering of terms in summations is different and S_1, S_2, S_3 and $event_S$ are substituted by T_1, T_2, T_3 and $event_T$, respectively.

Proposition 1 $\phi(F_1) \equiv \phi(G_1)$

Proof. $\phi(F_1) \equiv \phi(G_1)$ *as they can be converted to each other by renaming the process identifiers and reordering of terms in summations.*

Proposition 2 $F_1 \cong G_1$

Proof. By Proposition 1 and Theorem 5.

4.3 Strong Behavioural Equivalence

This section starts by presenting the name substitution function [77, 74, 87] and strong open bisimulation [110, 87, 102, 92]. Then a second type of equivalence of statechart diagrams called strong behavioural equivalence is defined.

Definition 20 *The name substitution function* $\sigma : \mathcal{N} \rightarrow \mathcal{N}$, *written* $\{\vec{x}/\vec{y}\}$, *replaces each* y_i *by* x_i *where* $i \in \mathbb{Z}^+$.

Definition 21 (Strong Open Bisimulation [87]) *A symmetric binary relation* $\mathcal{R}$ *on processes is a strong open bisimulation if* $(P, Q) \in \mathcal{R}$ *implies* $\forall \sigma$ *whenever* $P\sigma \xrightarrow{\alpha} P'$ *where* $bn(\alpha) \cap fn(P\sigma, Q\sigma) = \emptyset$ *then,* $\exists Q' : Q\sigma \xrightarrow{\alpha} Q' \wedge (P', Q') \in \mathcal{R}$. P *is strongly open bisimilar to* Q, *written* $P \dot{\sim}_o Q$, *if they are related by a strong open bisimulation.*

Two processes P and Q are strongly open bisimilar if they can simulate the behaviour of each other under all substitutions.

Next, we define a number of definitions which we will make use of in the definition of strong behavioural equivalence.

Let $S_1 \overset{t_1}{\rightharpoonup} S_2 \overset{t_2}{\rightharpoonup} \ldots \overset{t_{n-1}}{\rightharpoonup} S_n$ represents a trace which comprises a sequence of states and transitions where $S_1, S_2, \ldots, S_n \in \mathcal{ST}$ and $t_1, t_2, \ldots, t_{n-1} \in \mathcal{TR}$. We also let Θ be a set of traces.

Definition 22 *Suppose* $tr_1 = S_{1,1} \overset{t_{1,1}}{\rightharpoonup} S_{1,2} \overset{t_{1,2}}{\rightharpoonup} \ldots \overset{t_{1,n-1}}{\rightharpoonup} S_{1,n}$ *and* $tr_2 = S_{2,1} \overset{t_{2,1}}{\rightharpoonup} S_{2,2} \overset{t_{2,2}}{\rightharpoonup} \ldots \overset{t_{2,n-1}}{\rightharpoonup} S_{2,n}$. *The two traces* tr_1 *and* tr_2 *are identical, written* $tr_1 =_{TR} tr_2$, *iff* $\bigwedge_{i=1}^{n}(S_{1,i} = S_{2,i}) \wedge \bigwedge_{i=1}^{n-1}(t_{1,i} = t_{2,i})$ *holds.*

$S_{1,i} = S_{2,i}$ returns true if $S_{1,i}$ and $S_{2,i}$ represent the same state, whereas $t_{1,i} = t_{2,i}$ returns true if $t_{1,i}$ and $t_{2,i}$ denote the same transition.

Likewise, we let $S_1 \overset{E_1}{\rightharpoondown} S_2 \overset{E_2}{\rightharpoondown} \ldots \overset{E_{n-1}}{\rightharpoondown} S_n$ signifies a computation which consists of a sequence of states and events where $S_1, S_2, \ldots, S_n \in \mathcal{ST}$ and $E_1, E_2, \ldots, E_{n-1} \in \mathcal{E}$. The set of computations is denoted by Ψ.

Definition 23 *Given* $\psi_1 = S_{1,1} \overset{E_{1,1}}{\rightharpoonup} S_{1,2} \overset{E_{1,2}}{\rightharpoonup} \ldots \overset{E_{1,n-1}}{\rightharpoonup} S_{1,n}$ *with a corresponding trace* $tr_1 = S_{1,1} \overset{t_{1,1}}{\rightharpoonup} S_{1,2} \overset{t_{1,2}}{\rightharpoonup} \ldots \overset{t_{1,n-1}}{\rightharpoonup} S_{1,n}$ *and* $\psi_2 = S_{2,1} \overset{E_{2,1}}{\rightharpoonup} S_{2,2} \overset{E_{2,2}}{\rightharpoonup} \ldots \overset{E_{2,n-1}}{\rightharpoonup} S_{2,n}$ *with a corresponding trace* $tr_2 = S_{2,1} \overset{t_{2,1}}{\rightharpoonup} S_{2,2} \overset{t_{2,2}}{\rightharpoonup} \ldots \overset{t_{2,n-1}}{\rightharpoonup} S_{2,n}$. *The two computations are identical, written* $\psi_1 =_c \psi_2$, *iff* $\bigwedge_{i=1}^{n}(S_{1,i} = S_{2,i}) \wedge \bigwedge_{i=1}^{n-1}(E_{1,i} = E_{2,i})$ *holds.*

Definition 24 (Redundancy) *For any statechart diagram* F *which has traces* $tr_1, tr_2 \in \Theta$ *and corresponding computations* $\psi_1, \psi_2 \in \Psi$. *If* $\neg(tr_1 =_{TR} tr_2) \wedge \psi_1 =_c \psi_2$, *then* F *has a number of redundant states and transitions.*

Definition 25 (Strong Behavioural Equivalence) *Given two statechart diagrams* $F, G \in \mathcal{SC}$ *and* G *has a number of redundant states and transitions,* F *and* G *are strongly behavioural equivalent, written* $F \simeq G$, *if and only if the firing of every transition of* $F(G)$ *which consists of an event, a guard-condition and an action is matched by the firing of a transition of* $G(F)$ *with the same event, guard-condition and action.*

Theorem 7 *Given* $F, G \in \mathcal{SC}, F \simeq G \leftrightarrow \phi(F) \dot{\sim}_o \phi(G)$.
Proof sketch. $(\rightarrow)$ *Let* $S_1, S_2 \in States(F), S_1, S_2 \in States(G)$. *Suppose* $F \simeq G$, *a transition* t_1 *connects* S_1 *to* S_2 *in both* F *and* G *and a redundant transition* t_2 *with the same trigger and action as* t_1 *connects* S_1 *to a redundant state* S_2 *in* G. *Since* $F \simeq G$, F *and* G *have the same behaviour. Then the behaviour of processes defined by* $\phi_{state}(S_1)$ *of* F *and* $\phi_{state}(S_1)$ *of* G *is not discriminated by strong open bisimulation as they are both evolved to* $\phi_{state}(S_2)$ *through two identical sequences of reductions. Thus, if* $F \simeq G$ *then* $\phi(F) \dot{\sim}_o \phi(G)$.
$(\leftarrow)$ *Analogous to* $(\leftarrow)$ *of Theorem 5.*

Theorem 8 *The relation* $\simeq$ *is an equivalence.*
Proof. Analogous to Theorem 6.

Theorem 9 *Let* F *and* G *be any statechart diagrams. If* $F \cong G$ *then* $F \simeq G$.
Proof. Suppose $F \cong G$. *Since* $F \cong G$, *it follows that* $\phi(F) \equiv \phi(G)$. *Therefore,* $\phi(F) \dot{\sim}_o \phi(G)$. *Thus, if* $F \cong G$ *then* $F \simeq G$.

Theorem 9 states that isomorphism is stronger than strong behavioural equivalence. Consider the statechart diagrams F_2 and G_2 in Figure 4.2. Suppose that traces tr_1 and

tr_2 and their corresponding computations ψ_1 and ψ_2 are defined as follows:

$$tr_1 = T_1 \xrightarrow{t_1} T_2 \xrightarrow{t_2} T_1$$
$$tr_2 = T_1 \xrightarrow{t_3} T_2 \xrightarrow{t_4} T_1$$
$$\psi_1 = T_1 \xrightarrow{E_1} T_2 \xrightarrow{E_2} T_1$$
$$\psi_2 = T_1 \xrightarrow{E_1} T_2 \xrightarrow{E_2} T_1$$

The traces tr_1 and tr_2 are not identical as transition t_1 is different from transition t_3. A similar argument also holds for transitions t_2 and t_4. In contrast, the computations ψ_1 and ψ_2 are identical according to Definition 23. Hence, we do not discriminate the statechart diagram F_2 and G_2 as G_2 is the same as F_2 except that it has a redundant state T_2 and two redundant transitions labelled t_3 and t_4. The statechart diagrams F_2 and G_2 have a similar structure and exhibit the same behaviour. The upper and lower portions of statechart diagram G_2 are just mirror images of each other.

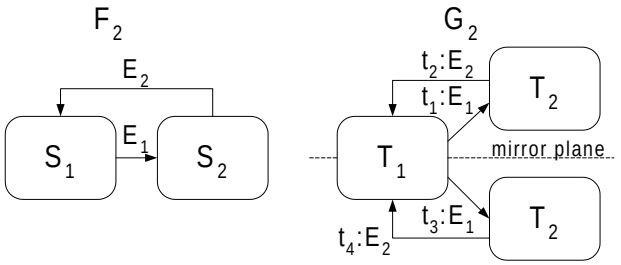

Figure 4.2: Example 2

We translate F_2 and G_2 into the π-calculus, construct the transition graphs for the π-calculus expressions and proceed to prove that they are strongly behavioural equivalent.

Figure 4.3 shows the transition graphs for statechart diagrams F_2 and G_2. The arrow denotes action, while the dotted arrow denotes name instantiation. To improve visual clarity, we exclude the output action *step* of the π-calculus expressions from the diagrams.

Proposition 3 $\phi(F_2) \dot{\sim}_o \phi(G_2)$

Proof sketch. Consider the transitions of S_1 *and* T_1, $S_1 \overset{event_S(x)}{\rightsquigarrow}$ *is simulated by* $T_1 \overset{event_T(x)}{\rightsquigarrow}$ (*Figure* 4.3).*Likewise,* $\overset{[x=e_1],event_S(x)}{\rightsquigarrow}$, $\overset{[x=e_i]}{\rightsquigarrow}$ S_1, $\overset{[x=e_2]}{\rightsquigarrow}$ S_1 *and* $\overset{[x=e_i]}{\rightsquigarrow}$ S_2

are matched by $\overset{[x=e_1],event_T(x)}{\rightsquigarrow}$, $\overset{[x=e_i]}{\rightsquigarrow}$ T_1, $\overset{[x=e_2]}{\rightsquigarrow}$ T_1 *and* $\overset{[x=e_i]}{\rightsquigarrow}$ T_2, *respectively. Similar arguments hold true in the opposite direction. Thus,* $\phi(F_2) \dot{\sim}_o \phi(G_2)$.

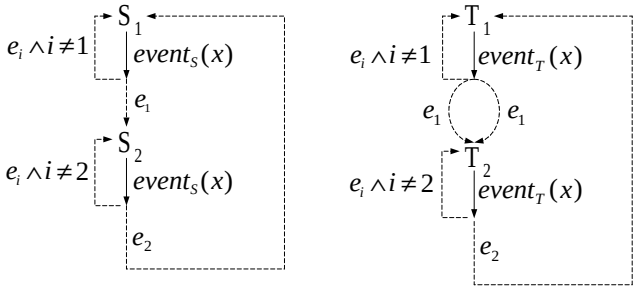

Figure 4.3: Transition graphs for Example 2

Proposition 4 $F_2 \simeq G_2$

Proof. By Proposition 3 and Theorem 7.

4.4 Weak Behavioural Equivalence

The notion of weak open bisimulation [110, 87, 102, 92] is used to define the third type of equivalence of statechart diagrams named weak behavioural equivalence. We classify statechart diagrams which are related by clustering and serialization as weakly behavioural equivalent. These statechart diagrams have different structure but exhibit the same behaviour.

Definition 26 (Weak Open Bisimulation [87]) *A symmetric binary relation* $\mathcal{R}$ *on processes is a weak open bisimulation if* $(P, Q) \in \mathcal{R}$ *implies* $\forall\sigma$ *whenever* $P\sigma \xrightarrow{\alpha} P'$ *where* $bn(\alpha) \cap fn(P\sigma, Q\sigma) = \emptyset$ *then,* $\exists Q' : Q\sigma \stackrel{\hat{\alpha}}{\Longrightarrow} Q' \wedge (P', Q') \in \mathcal{R}$. P *is weakly open bisimilar to* Q, *written* $P \dot{\approx}_o Q$, *if they are related by a weak open bisimulation.*

Unlike strong open bisimulation, weak open bisimulation does not differentiate between two π-calculus processes which differ from each other in sequences of internal actions which are unobservable.

Definition 27 (Clustering) *For any statechart diagram* F *which consists of states* $S_1, S_2, \ldots, S_n$. *If there exists a transition* $t \in \mathcal{TR}$ *from each of the states* $S_1, S_2, \ldots, S_{n-1}$

to S_n, $S_1, S_2, \ldots, S_{n-1}$ can be clustered by a composite state S'. A single transition t can be drawn from the edge of S' to S_n instead of having multiple transitions which originate from individual states $S_1, S_2, \ldots, S_{n-1}$.

Clustering transforms a flat statechart diagram into a hierarchical statechart diagrams. It reduces the number of transitions and improves visual clarity.

Definition 28 (Serialization) *Suppose F is a statechart diagram. If F consists of orthogonal regions $V_0, W_0, \ldots$ where $substates(V_0) = \{V_1, V_2, \ldots\}$ and $substates(W_0) = \{W_1, W_2, \ldots\}$, an equivalent statechart diagram F' can be constructed with states $substates(V_0) \times substates(W_0) \times \ldots$ which are cartesian products of the substates of the orthogonal regions. Statechart diagram F' is a serialized version of the statechart diagram F.*

Definition 29 (Weak Behavioural Equivalence) *Given two statechart diagrams F, $G \in \mathcal{SC}$ and they are related by clustering or serialization, F and G are weakly behavioural equivalent, written $F \approx G$, if and only if the firing of every transition of $F(G)$ which consists of an event, a guard-condition and an action is matched by the firing of a transition of $G(F)$ with the same event, guard-condition and action.*

Theorem 10 *Given $F, G \in \mathcal{SC}, F \approx G \leftrightarrow \phi(F) \dot{\approx}_o \phi(G)$.*
Proof sketch. ($\rightarrow$) Let F be a flat statechart diagram and G be the corresponding hierarchical statechart diagram. Suppose $F \approx G$ and they are related by clustering or serialization. Since $F \approx G$, F and G have the same behaviour. A transition which connects two non-composite states in F corresponds to a sequence of reductions without unobservable actions in $\phi(F)$ as shown in Case 1 of Theorem 1. A transition which involves a composite state in G relates to a sequence of reductions with unobservable actions in $\phi(G)$ as shown in Case 2 of Theorem 1. Therefore, this coincides with the definition of weak open bisimulation as the observable actions of the two sequences of reductions are equivalent. Thus, if $F \approx G$ then $\phi(F) \dot{\approx}_o \phi(G)$.
($\leftarrow$) Analogous to ($\leftarrow$) of Theorem 5.

Theorem 11 *The relation $\approx$ is an equivalence.*
Proof. Analogous to Theorem 6.

Theorem 12 *Let F and G be any statechart diagrams. If $F \simeq G$ then $F \approx G$.*
Proof. Analogous to Theorem 9.

Theorem 12 states that strong behavioural equivalence is finer than weak behavioural equivalence. Any statechart diagrams which are strongly behavioural equivalent are by implication also weakly behavioural equivalent.

Corollary 1 *Let F and G be any statechart diagrams. If $F \cong G$ then $F \approx G$.*
Proof. By Theorems 9 and 12.

Corollary 2 $\approx \subseteq \simeq \subseteq \cong$
Proof. Follows directly from Theorems 9 and 12.

Corollary 2 stipulates the inclusion relationships among the three proposed equivalences.

Since events E_1 and E_2 are distinct, statechart diagram G_3 (Figure 4.4) is just another way of representing F_3. It clusters states V_1 and V_2 by a composite state T_1. State S_1 corresponds to states T_1 and V_1, state S_2 corresponds to states T_1 and V_2 and state S_3 corresponds to state T_2.

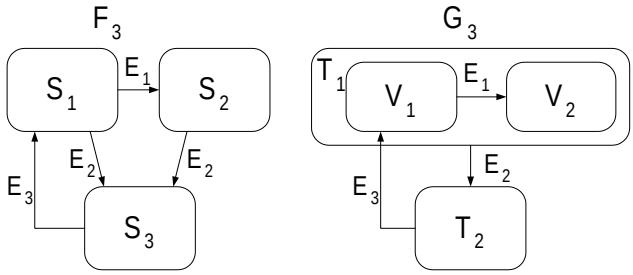

Figure 4.4: Example 3

Figure 4.5 shows the transition graphs for statechart diagrams F_3 and G_3. In Figure 4.5, the $\Longrightarrow$ arrow stands for one or more internal actions.

Proposition 5 $\phi(F_3) \,\dot{\approx}_o\, \phi(G_3)$
Proof sketch. Assume event channels e_1, e_2 and e_3 are distinct. Consider the transitions of S_1 and $T_1|V_1$, $S_1 \overset{event_S(x)}{\rightsquigarrow} $ is simulated by $T_1|V_1 \overset{event_T(x)}{\rightsquigarrow\rightsquigarrow}$. Following the same argument, $\overset{[x=e_1]}{\rightsquigarrow} S_2$, $\overset{[x=e_2]}{\rightsquigarrow} S_3$ and $\overset{[x=e_i]}{\rightsquigarrow} S_1$ are matched by $\overset{[x_1=e_1][x_2=neg],\tau}{\rightsquigarrow} T_1|V_2$, $\overset{[x_1=e_2][x_2=pos],\tau}{\rightsquigarrow}$ T_2 and $\overset{[x_1=e_i][x_2=neg],\tau}{\rightsquigarrow} T_1|V_1$. Likewise, $S_2 \overset{event_S(x)}{\rightsquigarrow}, S_3 \overset{event_S(x)}{\rightsquigarrow}, \overset{[x=e_2]}{\rightsquigarrow} S_3, \overset{[x=e_i]}{\rightsquigarrow} S_2, \overset{[x=e_3]}{\rightsquigarrow}$ S_1 and $\overset{[x=e_i]}{\rightsquigarrow} S_3$ are matched by $T_1|V_2 \overset{event_T(x)}{\rightsquigarrow\rightsquigarrow}, T_2 \overset{event_T(x)}{\rightsquigarrow}, \overset{[x_1=e_2][x_2=pos],\tau}{\rightsquigarrow} T_2, \overset{[x_1=e_i][x_2=neg],\tau}{\rightsquigarrow}$ $T_1|V_2, \overset{[x=e_3]}{\rightsquigarrow} T_1|V_1$ and $\overset{[x=e_i]}{\rightsquigarrow} T_2$. The converse holds true by similar arguments. Thus, $\phi(F_3) \,\dot{\approx}_o\, \phi(G_3)$.

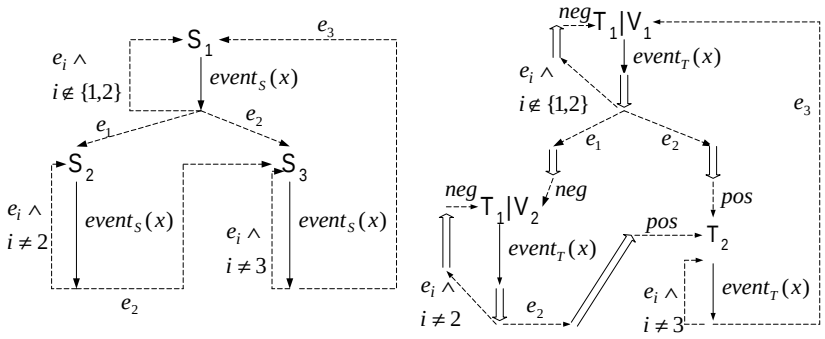

Figure 4.5: Transition graphs for Example 3

Proposition 6 $F_3 \approx G_3$

Proof. Proposition 5 and Theorem 10.

Differences in execution semantics may affect the equivalences of statechart diagrams as discussed in Section 2.4. If STATEMATE semantics [44] is used instead of UML semantics, F_3 is not equivalent to G_3 upon the occurrence of events E_1 and E_2. In UML semantics only one event is processed at a time, while in STATEMATE semantics a set of events is processed at a time. Non-determinism arises in F_3, whereas only the transition triggered by E_2 is fired in G_3 as STATEMATE has adopted an outer-first firing priority scheme.

Figure 4.6 shows another example of clustering which is based on [43]. The states V_1, V_2 and V_3 are clustered by a composite state S_1. The self-transition of S_1 means that each of its substates V_1, V_2 and V_3 has non-deterministic conflicting transitions. After the receipt of event E_1, the substate either proceeds as itself or evolves to one of the other two substates. The unfolded version of the statechart diagram (right hand diagram) simply exhibits the same behaviour in an explicit way. State T_1 corresponds to S_1 and V_1, state T_2 corresponds to S_1 and V_2 and state T_3 corresponds to S_1 and V_3.

Figure 4.7 depicts two models of parallelism. Statechart diagram F_4 uses an explicit parallelism model. Unlike statechart diagram F_4, statechart diagram G_4 uses an implicit parallelism model which is based on interleaving semantics. As defined in Definition 28, the states of G_4 are cartesian products of those substates in the orthogonal regions of F_4. States V_1 and W_1 are related to state T_2, states V_2 and W_1 are related to state T_3, states V_1 and W_2 are related to state T_4 and states V_2 and W_2 are related to state T_5. Statechart diagram G_4 is a serialized version of the statechart diagram F_4

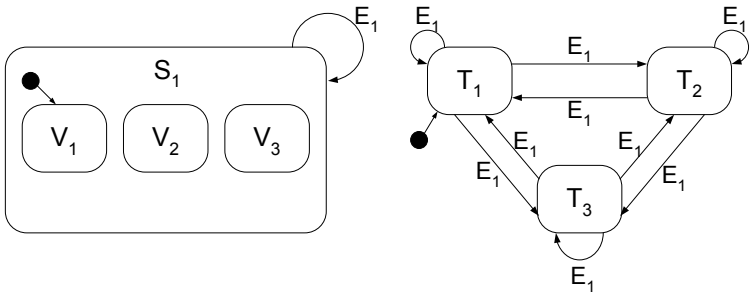

Figure 4.6: Example 4

if E_1 and E_2 are distinct.

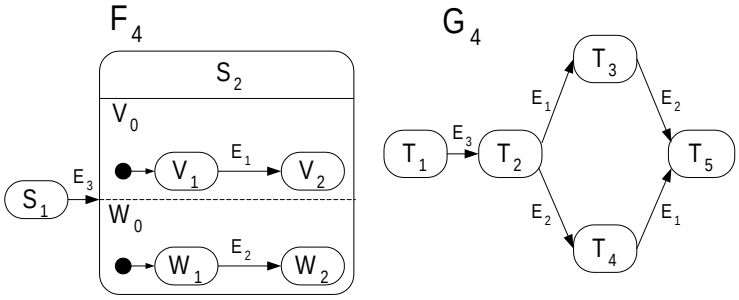

Figure 4.7: Example 5

Proposition 7 $\phi(F_4) \dot{\approx}_o \phi(G_4)$

Proof. Analogous to Proposition 5.

Proposition 8 $F_4 \approx G_4$

Proof. By Proposition 7 and Theorem 10.

A more complex example derived from [42] illustrating the two models of parallelism is shown in Figures 4.8 and 4.9. State T_1 corresponds to S_1, V_1 and W_1, state T_2 corresponds to S_1, V_1 and W_2, state T_3 corresponds to S_1, V_1 and W_3, state T_4 corresponds to S_1, V_2 and W_2, state T_5 corresponds to S_1, V_2 and W_3, state T_6 corresponds to S_1, V_2 and W_1.

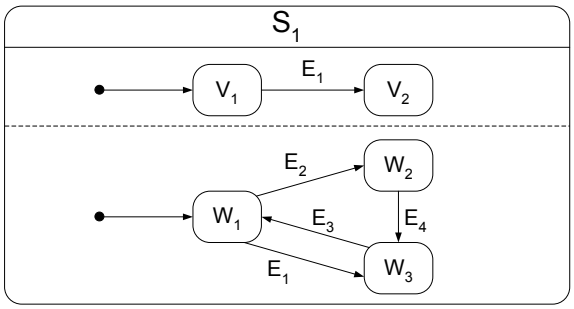

Figure 4.8: Example 6

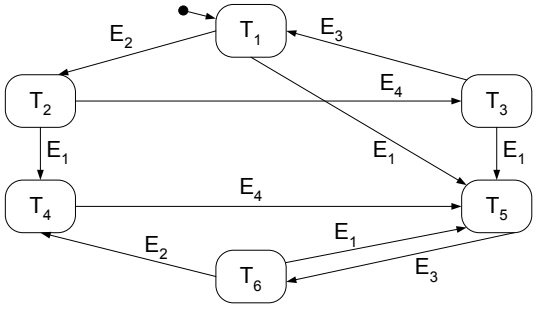

Figure 4.9: Example 6 (serialized version)

4.5 The Notion of Distinction

The strong open bisimulation and weak open bisimulation discussed in previous sections have two limitations. First, there is no way to specify formally that the event channels are distinct. In both Examples 3 and 5, the condition for the two pairs of statechart diagrams to be equivalent is that events E_1 and E_2 must not be equal. In general, different events should always be referred to by different event names and all event names should be distinct. In the π-calculus, this means that we need a way to represent when channels are distinct such that they are not substituted by the same channel. Second, all other channels other than the event channels which are parameters of a process definition should also be regarded as distinct since they represent completely different concepts according to the translation rules.

In this section, we overcome these limitations by introducing new open bisimulation definitions which are based on the previous ones. To specify that a pair of channels (names) is distinct, the notion of distinction [102] is defined as follows:

Definition 30 (Distinction [110, 111, 87, 102, 103]) *A distinction* D=$\{(x,y)|x,y \in \mathcal{N} \wedge (y,x) \in D \wedge x \neq y\}$ *is a finite symmetric and irreflexive binary relations on channels (names). A name substitution* σ *respects a distinction* D *if for all* $(x,y) \in D$ *it holds that* $\sigma(x) \neq \sigma(y)$.

Literally, this says a distinction is a set which contains pairs of distinct channels and no two channels which are required to be distinct are replaced by the same channel under a name substitution.

Definition 31 (Distinction-Indexed Set [110, 111, 87, 102, 103]) *A distinction-indexed set* $\mathcal{R} = \{\mathcal{R}_D|D \in Distinct\}$ *is a set of binary relations* $\mathcal{R}_D$ *on processes where* $Distinct$ *is a set of distinctions.*

In distinction-indexed set, each binary relation $\mathcal{R}_D \in \mathcal{R}$ is indexed by a unique distinction $D \in Distinct$.

Definition 32 (Strong Open Bisimulation [110, 111, 87, 102, 103]) *A distinction-indexed set* $\mathcal{R}$ *is a strong open bisimulation if* $\forall\sigma$ *which respect a distinction* $D \in Distinct$, $(P,\ Q) \in \mathcal{R}_D$ *and* $P\sigma \xrightarrow{\alpha} P'$ *where* $bn(\alpha) \cap fn(P\sigma, Q\sigma) = \emptyset$ *implies that (i) When* $\alpha = \overline{x}\boldsymbol{\nu}y$ *then* $\exists Q' : Q\sigma \xrightarrow{\overline{x}\nu y} Q' \wedge (P', Q') \in \mathcal{R}_{D'}$ *for* $D' = D\sigma \cup (\{y\} \times fn(P\sigma, Q\sigma)) \cup (fn(P\sigma, Q\sigma) \times \{y\})$*; and (ii) When* $\alpha \neq \overline{x}\boldsymbol{\nu}y$ *then* $\exists Q' : Q\sigma \xrightarrow{\alpha} Q' \wedge (P', Q') \in \mathcal{R}_{D\sigma}$. P *is strongly open D-bisimilar to* Q*, written* $P \dot{\sim}_o^D Q$*, if there is a distinction-indexed set* $\mathcal{R}$ *such that* $(P, Q) \in \mathcal{R}_D$.

Clause (i) states that if α is a bound output action i.e. $\overline{x}\boldsymbol{\nu}y$, the channel y is distinct from any free names of processes $P\sigma$ and $Q\sigma$. Clause (ii) specifies that if α is not a bound output action, the substituted channels are also required to be distinct. The strong open bisimulation defined earlier is actually equal to the Clause (ii) of the new definition when the distinction D is an empty set i.e. $P \dot{\sim}_o Q$ and $P \dot{\sim}_o^{\emptyset} Q$ are equivalent. The new weak open bisimulation is also defined in a similar manner using the notion of distinction.

4.6 Related Work

A similar study has been performed by Maggiolo-Schettini, Peron and Tini [68]. Statecharts are translated into labelled transition systems (LTS) using a variant of Pnueli

and Shalev semantics. Various types of equivalences are defined and their congruence properties are discussed. In [85], Park et al. have translated statecharts into Algebra of Communicating Shared Resources (ACSR) using STATEMATE semantics. Equivalence of statecharts is verified by illustrating that a bisimulation relationship does exist between them.

Our work is significantly different from their works as (i) we are studying UML statechart diagrams instead of Harel's statecharts; (ii) we have adopted UML semantics; (iii) we have developed a translator which is presented in Chapter 7 rather than performing the transformation only by hand; and (iv) both [68] and [85] fail to prove the equivalence of two statecharts when transitions cross borders of states (see Example 3) and we overcome this limitation using the π-calculus.

4.7 Summary

We have defined three types of equivalences: isomorphism, strong behavioural equivalence and weak behavioural equivalence. We have adopted open bisimulation rather than early and late bisimulations as strong open bisimulation is a congruence. It preserves all π-calculus operators. Two π-calculus processes representing statechart diagrams are congruent with respect to the π-calculus operators if they are strongly open bisimilar. As an example, the π-calculus representations of F_2 and G_2 (example 2) are congruent. In addition, the name instantiation of open bisimulation has adopted a call-by-need approach which also greatly reduces the number of substitutions and provides an efficient way for tool development.

An automated equivalence-checking environment for checking strong and weak behavioural equivalences of statechart diagrams which are based on our theoretical work in this chapter is a topic to be considered in Chapter 7.

Chapter 5

Symbolic Model Checking of Statechart Diagrams

Temporal logic [6, 49] is a formalism which specifies how the truth of a formula changes dynamically as time proceeds. In temporal logic there are two models of time. One is linear in which there is a single future time at any given point of time. The other is branching and at any given point of time there are multiple future times. The expressiveness of linear and branching temporal logics is different. There are some properties which can only be specified in branching temporal logic but not in linear temporal logic and vice versa. A detailed discussion which includes examples and proofs is given in [49, 28].

Computation Tree Logic (CTL) [49, 29, 30] is a branching temporal logic which was developed by Clarke and Emerson. It extends propositional logic by incorporating path quantifiers and temporal operators.

Symbolic model checking [30] is a formal verification technique which is based on ordered-binary decision diagrams (OBDDs) [20]. The correctness of a finite-state transition system is determined by checking whether the system satisfies specifications which are expressed as a number of properties in CTL. Symbolic Model Verifier (SMV) [71] is a symbolic model checker which automates the verification process. It returns a counterexample whenever the property being checked does not hold.

NuSMV [23] reimplements the SMV model checker. It extends and improves SMV in a number of areas [26, 27, 25] as summarized in the following three paragraphs.

First, new functionalities are added to NuSMV. SAT-based (propositional satisfia-

bility based) model checking [12] which is based on Reduced Boolean Circuit (RBC) representations supplements BDD-based (binary decision diagram based) model checking which is based on BDD representations. In contrast to SMV which supports only BDD-based model checking, NuSMV supports both BDD-based and SAT-based model checking. SAT-based model checking has advantages over BDD-based model checking as it uses much less space and does not encounter the state explosion problem. Another major enhancement is the interaction between the software tool and user. SMV runs in batch mode only. NuSMV, unlike SMV, operates in both batch and interactive modes.

Second, the performance of NuSMV is improved when compared with SMV. It addresses the state explosion problem in a more effective way. The detailed results of the performance tests appear in [27].

Third, NuSMV adopts the open source model [84] in which the source code is freely available and extensible. In NuSMV there is still further development on the software by various parties, whereas in SMV the development has actually stopped.

Following this line of research, this chapter puts forward a new approach by combining three threads, UML statechart diagrams, the π-calculus and the NuSMV model checker, as an approach to the specification, analysis and reasoning about finite state systems. The rest of the chapter is structured as follows. Sections 5.1 and 5.2 provide an overview of CTL and the NuSMV model checker. Section 5.3 examines the encoding of the UML statecharts based π-calculus in the NuSMV language. The correspondence between UML statecharts based π-calculus and NuSMV language is shown in Section 5.4. Section 5.5 describes prior work in the area. Section 5.6 concludes the chapter.

This chapter with the exception of Section 5.1 is an extended version of the material presented in [58].

5.1 Computation Tree Logic

In CTL, the traditional propositional logic is extended by incorporating path quantifiers and temporal operators. We let AP be a set of atomic propositions and let $p, p_1, p_2, \ldots, p_n$ range over AP.

Definition 33 (CTL Formulas [49]) *The syntax of CTL formulas is defined in Backus Naur form by:*

$\psi ::= p \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \psi \Rightarrow \psi \mid \mathbf{AX}\psi \mid \mathbf{AG}\psi \mid \mathbf{AF}\psi \mid \mathbf{A}[\psi \mathbf{U} \psi] \mid \mathbf{EX}\psi \mid \mathbf{EG}\psi \mid \mathbf{EF}\psi \mid \mathbf{E}[\psi \mathbf{U} \psi]$.

A and **E** are path quantifiers, while $\mathbf{X}, \mathbf{G}, \mathbf{F}$ and **U** are temporal operators. **A** means "for all paths" and **E** means "for some paths". $\mathbf{X}, \mathbf{G}$, **F** and **U** stand for "next", "globally", "future" and "until", respectively.

$\mathbf{AX}\psi$: means ψ is true in next state for all paths.

$\mathbf{AG}\psi$: means ψ is true in every future state for all paths.

$\mathbf{AF}\psi$: means ψ is true in some future state for all paths.

$\mathbf{A}[\psi_1 \mathbf{U} \psi_2]$: means ψ_1 is true until ψ_2 is true in some future state for all paths.

$\mathbf{EX}\psi$: means ψ is true in next state for some paths.

$\mathbf{EG}\psi$: means ψ is true in every future state for some paths.

$\mathbf{EF}\psi$: means ψ is true in some future state for some paths.

$\mathbf{E}[\psi_1 \mathbf{U} \psi_2]$: means ψ_1 is true until ψ_2 is true in some future state for some paths.

The desired properties of a system are classified into two main types: safety and liveness [59]. In general, a safety property expresses that a bad thing never occurs and a liveness property expresses that a good thing will ultimately occur. For instance, a safety property that specifies ψ_1 and ψ_2 will never occur simultaneously is written in CTL as:

$$\mathbf{AG}\neg(\psi_1 \wedge \psi_2)$$

A liveness property that specifies that if ψ_1 holds then ψ_2 will eventually occur is written in CTL as:

$$\mathbf{AG}(\psi_1 \Rightarrow \mathbf{AF}\psi_2)$$

5.2 The NuSMV Model Checker

NuSMV is a model checker for verifying the model of a finite state system against its specifications expressed in temporal logic. A NuSMV program specifies both the model and the system specification. It starts with a top-level module called *Main* with no parameters. The module *Main* basically comprises three sections: VAR, ASSIGN and SPEC.

NuSMV variables are declared under the VAR section. Each variable declaration statement is associated with either a Boolean, a bounded integer subrange, an enumerated data type, a user defined module or a bounded array of one of these four data types.

The ASSIGN declarations specify the initial values of NuSMV variables and the transition relation between the current and next values of NuSMV variables. The transition relation is presented as a number of next statements. Each option of the next statement comprises two parts: a precondition and an expression which defines next possible values of a NuSMV variable. Now consider the program fragment in Figure 5.1.

```
ASSIGN
init(state) := 0;
next(state) :=
case
  !state : 1;
  state  : 0;
esac;
```

Figure 5.1: Program fragment

The above program fragment specifies that the initial value of the variable *state* is 0. Depending on the current value of the variable *state*, the next value is either 0 or 1.

The system specification is expressed as a CTL formula which starts with the keyword SPEC. The NuSMV model checker determines whether the model satisfies the system specification. If not, a counterexample is given to illustrate why the specification is not satisfied.

Since the error trace of the counterexample contains all the variables defined in a NuSMV program as well as their corresponding values, it can be related back to the original UML statechart diagrams by focusing on a few key variables which keep track of the current active state and substates, validity of a guard-condition, occurrence of an event and execution of an action.

5.3 Implementation of the π-Calculus in NuSMV

The π-calculus is an action-based formalism in which the behaviour is defined by labelled transition systems (LTSs), whereas a NuSMV program is based on a state-based

formalism in which the behaviour is defined by Kripke structures. The implementation of UML statecharts based π-calculus expressions in the NuSMV input language is generalized to a problem of defining a mapping between a UML statecharts based labelled transition system and a Kripke structure. In contrast to the mapping of [33] which is only for non-parameterized actions, our mapping is for both non-parameterized and parameterized actions.

5.3.1 A Mapping between UML Statecharts Based LTSs and Kripke Structures

This subsection first gives the definitions of a UML statecharts based LTS and a Kripke structure. Then a mapping between them is formally defined.

Definition 34 (UML Statecharts Based Labelled Transition System) *A UML statecharts based labelled transition system (LTS) is a 5-tuple* $\mathcal{M}_\pi = (\Sigma_\pi, I_\pi, \mathcal{A}_\pi, \mathcal{C}_\pi, \Delta_\pi)$ *where*

- Σ_π *is a set of states;*
- $I_\pi \subseteq \Sigma_\pi$ *is the set of initial states;*
- $\mathcal{A}_\pi$ *is a set of actions such that* $\mathcal{A}_\pi = \mathcal{A}_{in} \cup \mathcal{A}_{out} \cup \{\tau\}$*;*
- $\mathcal{C}_\pi$ *is a set of conditions ranged over by* $N_1, \ldots, N_n$*; and*
- $\Delta_\pi \subseteq \Sigma_\pi \times \mathcal{A}_\pi \times \mathcal{C}_\pi \times \Sigma_\pi$ *is a transition relation between a current state and its successor states in which a transition is labelled with an action and a condition.*

Unlike other definitions of an LTS [76] in which an input or output action does not have any parameters, our definition supports parameterized actions. In addition, we also extend the transition relation to include conditions for representing the matching constructs of the π-calculus.

Definition 35 (Kripke Structure) *A Kripke structure over a set of atomic propositions* AP *is a 4-tuple* $\mathcal{M}_K = (\Sigma_K, I_K, \Delta_K, \mathcal{L}_K)$ *where*

- Σ_K *is a finite set of states;*
- $I_K \subseteq \Sigma_K$ *is the set of initial states;*
- $\Delta_K \subseteq \Sigma_K \times \Sigma_K$ *is a transition relation which relates each state with its successor states; and*
- $\mathcal{L}_K : \Sigma_K \rightarrow 2^{\mathsf{AP}}$ *is a function which returns the set of atomic propositions which holds in a state.*

Comparing the definitions of a UML statecharts based LTS and a Kripke structure, it is clear that the transition relation of a UML statecharts based LTS differs from the transition relation of a Kripke structure. In a UML statecharts based LTS, the action and condition of a transition are specified in an explicit way. In a Kripke structure, these are incorporated into either the source state of the transition as a Boolean expression or the target state of the transition as assignment statement(s). The mapping between UML statecharts based LTSs and Kripke structures is formally described as follows:

Definition 36 (UML Statecharts based LTS to Kripke Structure Mapping) *Given a UML statecharts based LTS* $\mathcal{M}_\pi = (\Sigma_\pi,\ I_\pi, \mathcal{A}_\pi, \mathcal{C}_\pi, \Delta_\pi)$*, the corresponding Kripke structure is defined as* $\mathcal{M}_K = (\Sigma_K, I_K, \Delta_K, \mathcal{L}_K)$ *satisfying the conditions:*

(i)

$$
\begin{aligned}
&\forall S_1, S_2, event(x), [x = e].S_1, S_2 \in \Sigma_\pi \\
&\quad \wedge \quad event(x) \in \mathcal{A}_\pi \\
&\quad \wedge \quad [x = e] \in \mathcal{C}_\pi \\
&\quad \wedge \quad (S_1, event(x), [x = e], S_2) \in \Delta_\pi \\
&\quad \Rightarrow \quad S_1 \cup \{event = e\}, S_2 \in \Sigma_K \\
&\quad \wedge \quad (S_1 \cup \{event = e\}, S_2) \in \Delta_K
\end{aligned}
$$

(ii)

$$
\begin{aligned}
&\forall S_1, S_2, event(x\ ack), [x = e].S_1, S_2 \in \Sigma_\pi \\
&\quad \wedge \quad event(x\ ack) \in \mathcal{A}_\pi \\
&\quad \wedge \quad [x = e] \in \mathcal{C}_\pi \\
&\quad \wedge \quad (S_1, event(x\ ack), [x = e], S_2) \in \Delta_\pi \\
&\quad \Rightarrow \quad S_1 \cup \{event = e\}, S_2 \in \Sigma_K \\
&\quad \wedge \quad (S_1 \cup \{event = e\}, S_2) \in \Delta_K
\end{aligned}
$$

(iii)

$$\begin{aligned}
&\forall S_1, S_2, T_1, T_2, ack(x), [x = pos], \overline{ack}\langle pos\rangle.S_1, S_2, T_1, T_2 \in \Sigma_\pi \\
&\quad \wedge \quad T_1 \in substates(S_1) \\
&\quad \wedge \quad T_2 \in substates(S_2) \\
&\quad \wedge \quad ack(x), \overline{ack}\langle pos\rangle \in \mathcal{A}_\pi \\
&\quad \wedge \quad [x = pos] \in \mathcal{C}_\pi \\
&\quad \wedge \quad (T_1, \overline{ack}\langle pos\rangle, -, T_2) \in \Delta_\pi \\
&\quad \wedge \quad (S_1, ack(x), [x = pos], S_2) \in \Delta_\pi \\
&\quad \Rightarrow \quad T_1, T_2 \cup \{ack = pos\}, S_1 \cup \{ack = pos\}, S_2 \in \Sigma_K \\
&\quad \wedge \quad (T_1, T_2 \cup \{ack = pos\}), (S_1 \cup \{ack = pos\}, S_2) \in \Delta_K
\end{aligned}$$

(iv)

$$\begin{aligned}
&\forall S_1, S_2, \overline{ins}\langle e\rangle.S_1, S_2 \in \Sigma_\pi \\
&\quad \wedge \quad \overline{ins}\langle e\rangle \in \mathcal{A}_\pi \\
&\quad \wedge \quad (S_1, \overline{ins}\langle e\rangle, -, S_2) \in \Delta_\pi \\
&\quad \Rightarrow \quad S_1 \cup \{!ins\}, S_2 \cup \{ins = 1, buff = e\} \in \Sigma_K \\
&\quad \wedge \quad (S_1 \cup \{!ins\}, S_2 \cup \{ins = 1, buff = e\}) \in \Delta_K
\end{aligned}$$

(v)

$$\begin{aligned}
&\forall S_1, S_2, \overline{step}.S_1, S_2 \in \Sigma_\pi \\
&\quad \wedge \quad \overline{step} \in \mathcal{A}_\pi \\
&\quad \wedge \quad (S_1, \overline{step}, -, S_2) \in \Delta_\pi \\
&\quad \Rightarrow \quad S_1 \cup \{!step\}, S_2 \cup \{step = 1\} \in \Sigma_K \\
&\quad \wedge \quad (S_1 \cup \{!step\}, S_2 \cup \{step = 1\}) \in \Delta_K
\end{aligned}$$

where "–" denotes a transition relation with no condition and buff is a variable.

Conditions (i) and (ii) define that a UML statecharts based LTS is transformed into a Kripke structure by mapping each transition labelled with an input action and a matching construct to a transition which contains a corresponding Boolean expression in the current state. An input action and a matching construct are modelled as

a Boolean expression for indicating the input action is performed and the matching construct holds.

Condition (iii) specifies that a transition labelled with an output action and a transition labelled with an input action and a matching construct are mapped to two transitions. The former one contains a corresponding assignment in the next state, while the latter one contains a corresponding Boolean expression in the current state. The effect of an output action $\overline{ack}\langle pos\rangle$ is modelled by assigning a variable which represents the channel *ack* to a new value which represents the channel *pos*.

Condition (iv) states that a parameterized output action is transformed into a negated Boolean expression in the current state and two assignments in the next state. The negated Boolean expression is a precondition to ensure that the parameterized output action is performed only if it has not yet carried out. The first assignment sets the value of the Boolean variable to 1 (true) for representing the output action has performed, while the second assignment stores the parameter in the scalar variable.

Condition (v) stipulates that a non-parameterized output action is translated into a negated Boolean expression in the current state and an assignment statement in the next state. The assignment statement assigns the value 1 (true) to the Boolean variable. Unlike Condition (iv), there is not a second assignment as the output action does not have a parameter.

5.3.2 Encoding the UML Statecharts Based π-Calculus in NuSMV

The implementation of the UML statecharts based π-calculus in the NuSMV input language is defined formally as a set of rules which are based on the UML statecharts based LTS to Kripke structure mapping. In the following listings, italic font is used for π-calculus expressions and typewriter font is used for NuSMV code. In addition, an underline denotes the part of a π-calculus expression considered during a translation.

Rule 1 *A statechart diagram $F \in \mathcal{SC}$, in which each of its state is modelled by a process identifier in the π-calculus, is implemented in NuSMV as a module:*

```
MODULE F
```

Rule 1 states that a group of process identifiers which models the behaviour of a statechart diagram is declared under a module.

Rule 2 *Process identifiers* $A_i(\overrightarrow{x_i}) \in \Im$ *for* $i = 1, \ldots, n$ *representing direct substates of the root state of a statechart diagram are encoded in the NuSMV input language as a scalar variable state in which* A_i *for* $i = 1, \ldots, n$ *are its symbolic values.*

```
VAR
state: {A_1, A_2, ..., A_n};
```

Rule 3 *A null process* **0** *in the* π*-calculus is mapped to a symbolic value nil in NuSMV.*

Rule 4 *Process identifiers* $A_i(\overrightarrow{x_i}) \in \Im$ *for* $i = 1, \ldots, n$ *representing direct substates of all non-concurrent composite states of the root state of a statechart diagram are modelled in the NuSMV input language as a scalar variable substate in which* A_i *for* $i = 1, \ldots, n$ *and nil are its symbolic values.*

```
VAR
substate: {A_1, A_2, ..., A_n, nil};
```

Rule 5 *A channel* $e \in \mathcal{N}$ *representing an event* $E \in \mathcal{E}$ *in a statechart diagram is modelled as a symbolic value.*

Rules 2 and 4 specify that process identifiers are modelled in the NuSMV input language as scalar variables and symbolic values under the VAR section. The translation of an ordinary non-composite state is distinguished from a root state as the root state does not have any superstates and cannot have any outgoing transitions (Rule 9 of Chapter 3). A null process and an event channel are both translated into symbolic values as defined by Rules 3 and 5.

Rule 6 *An input action* $event(x) \in \mathcal{A}_{in}$ *and a matching construct* $[x = e_1]$ *which model the receipt of an event* $E_1 \in \mathcal{E}$ *are specified as a Boolean expression* $event\_buff = e_1$*.*

$$\underline{A(\vec{x}) \stackrel{\text{def}}{=} event(x).([x = e_1]\overline{step}.\underline{B(\vec{x})} + \Sigma_{i \neq 1}[x = e_i]\overline{step}.A(\vec{x}))}$$

```
next(state) :=
case
  state=A & event_buff=e_1 : B;
esac;
```

Rule 6 is derived from Condition(i) of Definition 36. The scalar variable *event_buff* stores the dispatched event which is chosen from the event queue of the statechart diagram. We use *event_buff* instead of *event* to improve readability of the generated code by emphasizing that the event is stored in a buffer represented as a variable in NuSMV.

Rule 7 *An output action* $\overline{step} \in \mathcal{A}_{out}$ *which models the step semantics is transformed into a Boolean variable step and a condition* !*step. The value* 1 *(true) signifies the end of a run-to-completion step, while the value* 0 *(false) means that a processing of an event is in progress.*

$$\underline{A(\vec{x}) \stackrel{\text{def}}{=} event(x).([x = e_1]\overline{step}.B(\vec{x}) + \Sigma_{i\neq 1}[x = e_i]\overline{step}.A(\vec{x}))}$$

```
next(step) :=
case
  state=A & event_buff=e_1 & !step : 1;
  event_buff!=empty & step         : 0;
esac;
```

The value of *step* is assigned to 0 whenever an event is waiting for executing and no processing of event is in progress.

Rule 8 *Given a process identifier* $A(\vec{x}) \in \Im$ *defining a process which evolves to itself (i.e. an implicit consumption of an event) is mapped to the default cases of next statements for variables other than the Boolean variable step.*

$$\underline{A(\vec{x})} \stackrel{\text{def}}{=} event(x).([x = e_1]\overline{step}.B(\vec{x}) + \underline{\Sigma_{i\neq 1}[x = e_i]\overline{step}.A(\vec{x})})$$

```
next(var) :=
case
  1    : var;
esac;
```

where var represents any variables other than step.

As an example, if *var* stands for *state*, the NuSMV code is represented as:

```
next(state) :=
case
  1    : state;
esac;
```

Rule 9 *For the Boolean variable step, the implicit consumption is implemented as:*

```
next(step) :=
case
  state=next(state) & !step : 1;
esac;
```

Rule 7 is derived from Condition (v) of the UML statecharts based LTS to Kripke structure mapping. The default cases of the next statements in Rule 8 define that the values of the variables remain unchanged. The condition $state = next(state)$ in Rule 9 checks whether the process evolves to itself. If so, the new value of *step* is set to 1 to indicate the end of a run-to-completion step.

Rule 10 *An output action $\overline{ins_F}\langle e_2\rangle \in \mathcal{A}_{out}$, which denotes the insertion of an event represented as e_2 to the event queue of a statechart diagram $F \in \mathcal{SC}$ (i.e. a send action), is expressed in the NuSMV language as a Boolean variable ins_F and a scalar variable F_q_buff. The scalar variable F_q_buff takes on e_2 as one of its possible symbolic values.*

$$\underline{A(\vec{x}) \stackrel{\text{def}}{=}}$$
$$\underline{event(x).([x = e_1]\overline{ins_F}\langle e_2\rangle.\overline{step}.B(\vec{x})} +$$
$$\Sigma_{i\neq 1}[x = e_i]\overline{step}.A(\vec{x}))$$

```
next(ins_F) :=
case
  state=A & event_buff=e_1 & !ins_F & !step : 1;
esac;

next(F_q_buff) :=
case
  state=A & event_buff=e_1 & !ins_F & !step : e_2;
esac;

next(state) :=
case
  state=A & event_buff=e_1 & !ins_F & !step : B;
esac;

next(step) :=
```

```
case
  state=A & event_buff=e_1 & !ins_F & !step : 1;
  state=A & event_buff=e_1 & ins_F & !step  : step;
esac;
```

The definition of Rule 10 is based on Condition (iv) of the UML statecharts based LTS to Kripke structure mapping. The insertion of e_2 to the event queue of F is regarded as setting the values of *ins_F* and *F_q_buff* to 1 (*true*) and e_2, respectively.

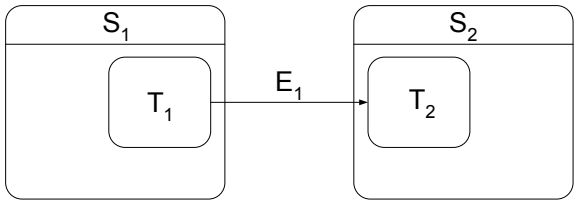

Figure 5.2: An interlevel transition

Rule 11 *Given* $S_1(step, event, \vec{e}, event_{substate1}, pos, neg)$, $S_2(step, event, \vec{e}, event_{substate2}, pos, neg)$, $T_1(event_{substate1}, \vec{e}, pos, neg)$, $T_2(event_{substate2}, \vec{e}, pos, neg) \in \Im$, $\phi^{-1}_{state}(S_1(step, event, \vec{e}, event_{substate1}, pos, neg))$, $\phi^{-1}_{state}(S_2(step, event, \vec{e}, event_{substate2}, pos, neg))$, $\phi^{-1}_{state}(T_1(event_{substate1}, \vec{e}, pos, neg))$, $\phi^{-1}_{state}(T_2(event_{substate2}, \vec{e}, pos, neg)) \in \mathcal{ST}$, $T_1 \in substates(S_1)$, $T_2 \in substates(S_2)$ *and an interlevel transition exits* T_1 *and enters* T_2 *directly when an event* E_1 *occurs (Figure 5.2). The transformation is defined by:*

(i) The input action $event_{substate1}(x_1\ ack) \in \mathcal{A}_{in}$ *and the matching construct* $[x_1 = e_1]$ *of* $T_1(event_{substate1}, \vec{e}, pos, neg)$ *are translated into* $event\_buff = e\_1$ *according to Condition (ii) of Definition 36.*

(ii) The channel ack is encoded in NuSMV as a scalar variable ack which takes symbolic values pos, neg *and undefine. The implementation of implicit consumption for the variable ack is defined by Rule 9. The condition* $!substate = nil$ *is added to the option of the case statement so that no negative acknowledgement is sent out when there is not any active substate (i.e. the value of the substate equals nil).*

(iii) The output action $\overline{ack}\langle pos\rangle \in \mathcal{A}_{out}$ *of* $T_1(event_{substate1}, \vec{e}, pos, neg)$, *input action* $ack(x_2) \in \mathcal{A}_{in}$ *and matching construct* $[x_2 = pos]$ *of* $S_1(step, event, \vec{e}, event_{substate1}, pos, neg)$, *by Condition (iii) of Definition 36, are implemented as conditions* $state = S\_1$ *and* $ack = pos$ *in the case statements of variables* $state, step$ *and*

substate and an option in the case statement of variable ack which its value is set to pos.

(iv) The condition $!superstate\_enable$ *is added to all options of case statements which model behaviour of substates, whereas the condition* $superstate\_enable$ *is added to all options of case statements which model behaviour of non-concurrent composite states.*

$$
\begin{aligned}
&S_1(step, event, \vec{e}, event_{substate1}, pos, neg) \stackrel{\text{def}}{=} \\
&\quad event(x_1).(\boldsymbol{\nu} ack)\overline{event_{substate1}}\langle x_1\ ack\rangle.ack(x_2). \\
&\quad ([x_2 = pos]\overline{step}.(\boldsymbol{\nu} event_{substate2}) \\
&\quad (S_2(step, event, \vec{e}, event_{substate2}, pos, neg)|T_2(event_{substate2}, \vec{e}, pos, neg)) + \\
&\quad [x_2 = neg]\overline{step}.S_1(step, event, \vec{e}, event_{substate1}, pos, neg))
\end{aligned}
$$

$$
\begin{aligned}
&T_1(event_{substate1}, \vec{e}, pos, neg) \stackrel{\text{def}}{=} \\
&\quad event_{substate1}(x_1\ ack).([x_1 = e_1]\overline{ack}\langle pos\rangle.\mathbf{0} + \\
&\quad \Sigma_{i\neq 1}[x_1 = e_i]\overline{ack}\langle neg\rangle.T_1(event_{substate1}, \vec{e}, pos, neg))
\end{aligned}
$$

```
next(state) :=
case
  state=S_1 & ack=pos & !step & superstate_enable     : S_2;
  1                                                    : state;
esac;

next(step) :=
case
  state=S_1 & ack=pos & !step & superstate_enable     : 1;
  event_buff!=empty & step                             : 0;
  1                                                    : step;
esac;

next(substate) :=
case
  state=S_1 & ack=pos & !step & superstate_enable   : T_2;
  substate=T_1 & event_buff=e_1 & !superstate_enable: nil;
  1                                                 : substate;
esac;
```

```
next(ack) :=
case
  substate=T_1 & event_buff=e_1 & !superstate_enable  : pos;
  substate=next(substate) & !step & !superstate_enable
  & !substate=nil                                     : neg;
  1                                                   : undefine;
esac;

init(superstate_enable) := 0;
next(superstate_enable) :=
case
  (next(ack)=pos | next(ack)=neg) & !step             : 1;
  1                                                   : 0;
esac;
```

The output actions $\overline{ack}\langle pos\rangle$ (positive acknowledgement) and $\overline{ack}\langle neg\rangle$ (negative acknowledgement) in process definition of $T_1(event_{substate1}, \vec{e}, pos, neg)$ and Boolean variable *superstate_enable* model the lower-first firing priority scheme. The default case sets the value of *ack* to $undefine$ for representing that neither *pos* nor *neg* is returned when there is no substate. In addition, the value of *ack* also remains undefined as long as the substate is waiting for the receipt of an event.

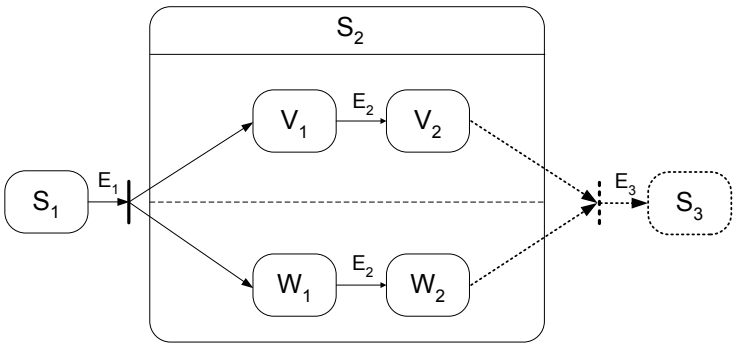

Figure 5.3: A fork

Rule 12 *Given* $\phi_{state}(S_1) = S_1(step, event_S, \vec{e}, \widetilde{ack})$, $\phi_{state}(S_2) = S_2(step, event_S, \vec{e}, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn})$, $\phi_{state}(V_1) = V_1\ (event_{substate1}, \vec{e}, \widetilde{ack}, cont_{substate1}, end_{substate1})$ *and* $\phi_{state}(W_1) = W_1(event_{substate2}, \vec{e}, \widetilde{ack}, cont_{substate2}, end_{substate2})$ *where* $\widetilde{ack} = pos, neg$ *and* $\widetilde{syn} = cont_{substate1}, end_{substate1}, cont_{substate2}, end_{substate2}$. *A split of*

control (the portion of Figure 5.3 which is drawn using solid lines and includes S_1, S_2, V_1 and W_1) is transformed into the NuSMV language using the same approach as Rule 11 with the exception of:

(i) The condition $ack = pos$ is deleted from the case statements of state, step, substate1 and substate2 as S_1 is a non-composite state.

(ii) The condition superstate_enable or !superstate_enable is removed from the case statements of state, step, substate1 and substate2 and the condition cont_substate1, cont_substate2, !cont_substate1 or !cont_substate2 is added to the case statements of substate1, substate2, ack_substate1 and ack_substate2, respectively, as the interaction between the composite state S_2 and its active substates V_1 and W_1 is based on the continuation signals $cont_{substate1}$ and $cont_{substate2}$. The conditions !cont_substate1 in the case statement of ack_substate1 and !cont_substate2 in the case statement of ack_substate2 ensure that the negative acknowledgement is sent before the continuation signal is generated.

(iii) The signals $cont_{substate1}, end_{substate1}, cont_{substate2}$ and $end_{substate2}$ are encoded in NuSMV as Boolean variables. The π-calculus statements $ack_{substate1}(x_2).ack_{substate2}(x_3)$ and $[x_2 = pos][x_3 = pos]\overline{end_{substate1}}$ are modelled as conditions ack_substate1=pos & ack_substate2=pos and the setting of end_substate1 to 1.

(iv) The case statement of superstate_enable is modified since there are more than one active substates.

$$
\begin{aligned}
&S_1(step, event_S, \vec{e}, \widetilde{ack}) \stackrel{\text{def}}{=} \\
&\quad event_S(x). \\
&\quad ([x = e_1]\overline{step}. \\
&\quad\ (\boldsymbol{\nu} event_{substate1}\ event_{substate2}\ cont_{substate1}\ end_{substate1} \\
&\quad\ \ cont_{substate2}\ end_{substate2}) \\
&\quad\ (S_2(step, event_S, \vec{e}, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn}) | \\
&\quad\ \ V_1(event_{substate1}, \vec{e}, \widetilde{ack}, cont_{substate1}, end_{substate1}) | \\
&\quad\ \ W_1(event_{substate2}, \vec{e}, \widetilde{ack}, cont_{substate2}, end_{substate2})) + \\
&\quad\ \Sigma_{i \neq 1}[x = e_i]\overline{step}.S_1(step, event_S, \vec{e}, \widetilde{ack}))
\end{aligned}
$$

$$
\begin{aligned}
&S_2(step, event_S, \vec{e}, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn}) \stackrel{\text{def}}{=} \\
&\quad event_S(x_1). \\
&\quad (\boldsymbol{\nu} ack_{substate1}\ ack_{substate2})\overline{event_{substate1}}\langle x_1\ ack_{substate1}\rangle. \\
&\quad \overline{event_{substate2}}\langle x_1\ ack_{substate2}\rangle.ack_{substate1}(x_2).ack_{substate2}(x_3). \\
&\quad ([x_2 = pos][x_3 = pos]\overline{end_{substate1}}.\overline{end_{substate2}}. \\
&\quad \overline{step}.S_3(step, event_S, \vec{e}, \widetilde{ack}) + \\
&\quad [x_2 = neg][x_3 = neg]\overline{cont_{substate1}}.\overline{cont_{substate2}}. \\
&\quad \overline{step}.S_2(step, event_S, \vec{e}, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn}) + \\
&\quad [x_2 = pos][x_3 = neg]\overline{cont_{substate1}}.\overline{cont_{substate2}}. \\
&\quad \overline{step}.S_2(step, event_S, \vec{e}, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn}) + \\
&\quad [x_2 = neg][x_3 = pos]\overline{cont_{substate1}}.\overline{cont_{substate2}}. \\
&\quad \overline{step}.S_2(step, event_S, \vec{e}, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn}))
\end{aligned}
$$

$$
\begin{aligned}
&V_1(event_{substate1}, \vec{e}, \widetilde{ack}, cont_{substate1}, end_{substate1}) \stackrel{\text{def}}{=} \\
&\quad event_{substate1}(x\ ack_{substate1}). \\
&\quad ([x = e_2]\overline{ack_{substate1}}\langle neg\rangle.cont_{substate1}. \\
&\quad V_2(event_{substate1}, \vec{e}, \widetilde{ack}, cont_{substate1}, end_{substate1}) + \\
&\quad \Sigma_{i\neq 2}[x = e_i]\overline{ack_{substate1}}\langle neg\rangle.cont_{substate1}. \\
&\quad V_1(event_{substate1}, \vec{e}, \widetilde{ack}, cont_{substate1}, end_{substate1}))
\end{aligned}
$$

$$
\begin{aligned}
&W_1(event_{substate2}, \vec{e}, \widetilde{ack}, cont_{substate2}, end_{substate2}) \stackrel{\text{def}}{=} \\
&\quad event_{substate2}(x\ ack_{substate2}). \\
&\quad ([x = e_2]\overline{ack_{substate2}}\langle neg\rangle.cont_{substate2}. \\
&\quad W_2(event_{substate2}, \vec{e}, \widetilde{ack}, cont_{substate2}, end_{substate2}) + \\
&\quad \Sigma_{i\neq 2}[x = e_i]\overline{ack_{substate2}}\langle neg\rangle.cont_{substate2}. \\
&\quad W_1(event_{substate2}, \vec{e}, \widetilde{ack}, cont_{substate2}, end_{substate2}))
\end{aligned}
$$

```
next(state) :=
  case
    state = S_1 & event_buff = e_1 & !step                : S_2;
    state = S_2 & end_substate1 & end_substate2 & !step    : S_3;
```

```
    state = S_2 & cont_substate1 & cont_substate2 & !step  : S_2;
    1                                                      : state;
  esac;
next(step) :=
  case
    state = S_1 & event_buff = e_1 & !step                 : 1;
    state = S_2 & end_substate1 & end_substate2 & !step    : 1;
    state = S_2 & cont_substate1 & cont_substate2 & !step  : 1;
    event_buff != empty & step                             : 0;
    1                                                      : step;
  esac;
next(substate1) :=
  case
    state = S_1 & event_buff = e_1 & !step                 : V_1;
    substate1 = V_1 & event_buff = e_2 & cont_substate1    : V_2;
    1                                                      : substate1;
  esac;
next(substate2) :=
  case
    state = S_1 & event_buff = e_1 & !step                 : W_1;
    substate2 = W_1 & event_buff = e_2 & cont_substate2    : W_2;
    1                                                      : substate2;
  esac;
next(ack_substate1) :=
  case
    substate1 = V_1 & event_buff = e_2 & !cont_substate1 &
    !superstate_enable                                     : neg;
    substate1 = next(substate1) & !step &
    !superstate_enable & !substate1 = nil                  : neg;
    1                                                      : undefine;
  esac;
next(ack_substate2) :=
  case
    substate2 = W_1 & event_buff = e_2 & !cont_substate2 &
    !superstate_enable                                     : neg;
    substate2 = next(substate2) & !step &
    !superstate_enable & !substate2 = nil                  : neg;
    1                                                      : undefine;
```

```
    esac;
  init(cont_substate1) := 0;
  next(cont_substate1) :=
    case
      state = S_2 & ack_substate1 = neg & ack_substate2 = neg
      & !step & superstate_enable                              : 1;
      state = S_2 & ack_substate1 = pos & ack_substate2 = neg
      & !step & superstate_enable                              : 1;
      state = S_2 & ack_substate1 = neg & ack_substate2 = pos
      & !step & superstate_enable                              : 1;
      state = S_2 & cont_substate1 & cont_substate2 & !step    : 0;
      1                                                        : cont_substate1;
    esac;
  init(cont_substate2) := 0;
  next(cont_substate2) :=
    case
      state = S_2 & ack_substate1 = neg & ack_substate2 = neg
      & !step & superstate_enable                              : 1;
      state = S_2 & ack_substate1 = pos & ack_substate2 = neg
      & !step & superstate_enable                              : 1;
      state = S_2 & ack_substate1 = neg & ack_substate2 = pos
      & !step & superstate_enable                              : 1;
      state = S_2 & cont_substate1 & cont_substate2 & !step    : 0;
      1                                                        : cont_substate2;
    esac;
  init(end_substate1) := 0;
  next(end_substate1) :=
    case
      state = S_2 & ack_substate1 = pos & ack_substate2 = pos
      & !step & superstate_enable                              : 1;
      state = S_2 & end_substate1 & end_substate2 & !step      : 0;
      1                                                        : end_substate1;
    esac;
  init(end_substate2) := 0;
  next(end_substate2) :=
    case
      state = S_2 & ack_substate1 = pos & ack_substate2 = pos
      & !step & superstate_enable                              : 1;
```

```
      state = S_2 & end_substate1 & end_substate2 & !step    : 0;
      1                                                      : end_substate2;
    esac;
  init(superstate_enable) := 0;
  next(superstate_enable) :=
    case
      (next(ack_substate1) = pos |
       next(ack_substate1) = neg) &
      (next(ack_substate2) = pos |
       next(ack_substate2) = neg) & !step                    : 1;
      1                                                      : 0;
    esac;
```

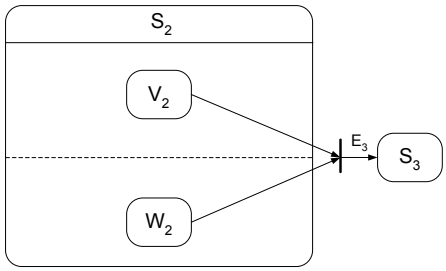

Figure 5.4: A join

Rule 13 *Given* $\phi_{state}(S_2) = S_2(step, event_S, \vec{e}, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn})$, $\phi_{state}(V_2) = V_2(event_{substate1}, \vec{e}, \widetilde{ack}, cont_{substate1}, end_{substate1})$, $\phi_{state}(W_2) = W_2(event_{substate2}, \vec{e}, \widetilde{ack}, cont_{substate2}, end_{substate2})$ *and* $\phi_{state}(S_3) = S_3(step, event_S, \vec{e}, \widetilde{ack})$ *where* $\widetilde{ack} = pos, neg$ *and* $\widetilde{syn} = cont_{substate1}, end_{substate1}, cont_{substate2}, end_{substate2}$. *A synchronization of control (Figure 5.4) is translated into NuSMV using the same approach as Rule 12. The additional translation clauses are defined by:*

(i) The encoding of process S_2 *is the same as the one defined in Rule 12.*

(ii) The conditions !cont_substate1 | !end_substate1 and !cont_substate2 | !end_substate2 are added to the case statements of ack_substate1 and ack_substate2 to ensure that the acknowledgement is sent before the continuation or termination signal is generated.

$$V_2(event_{substate1}, \vec{e}, \widetilde{ack}, cont_{substate1}, end_{substate1}) \stackrel{\text{def}}{=}$$
$$event_{substate1}(x\ ack_{substate1}).$$
$$([x = e_3]\overline{ack_{substate1}}\langle pos\rangle.$$
$$(end_{substate1} +$$
$$cont_{substate1}.V_2(event_{substate1}, \vec{e}, \widetilde{ack}, cont_{substate1}, end_{substate1})) +$$
$$\Sigma_{i\neq 3}[x = e_i]\overline{ack_{substate1}}\langle neg\rangle.cont_{substate1}.$$
$$V_2(event_{substate1}, \vec{e}, \widetilde{ack}, cont_{substate1}, end_{substate1}))$$

$$W_2(event_{substate2}, \vec{e}, \widetilde{ack}, cont_{substate2}, end_{substate2}) \stackrel{\text{def}}{=}$$
$$event_{substate2}(x\ ack_{substate2}).$$
$$([x = e_3]\overline{ack_{substate2}}\langle pos\rangle.$$
$$(end_{substate2} +$$
$$cont_{substate2}.W_2(event_{substate2}, \vec{e}, \widetilde{ack}, cont_{substate2}, end_{substate2})) +$$
$$\Sigma_{i\neq 3}[x = e_i]\overline{ack_{substate2}}\langle neg\rangle.cont_{substate2}.$$
$$W_2(event_{substate2}, \vec{e}, \widetilde{ack}, cont_{substate2}, end_{substate2}))$$

```
next(substate1) :=
  case
    substate1 = V_2 & event_buff = e_3 & end_substate1     : nil;
    substate1 = V_2 & event_buff = e_3 & cont_substate1    : V_2;
  esac;
next(substate2) :=
  case
    substate2 = W_2 & event_buff = e_3 & end_substate2     : nil;
    substate2 = W_2 & event_buff = e_3 & cont_substate2    : W_2;
  esac;
next(ack_substate1) :=
  case
    substate1 = V_2 & event_buff = e_3 & (!cont_substate1 |
    !end_substate1) & !superstate_enable                   : pos;
  esac;
next(ack_substate2) :=
  case
    substate2 = W_2 & event_buff = e_3 & (!cont_substate2 |
```

```
    !end_substate2) & !superstate_enable                    : pos;
esac;
```

The representations of a concurrent composite state with a fork pseudostate and a join pseudostate are defined by Rules 12 and 13. The variables $cont_{substate1}$, $end_{substate1}$, $cont_{substate2}$ and $end_{substate2}$ implemented as Boolean variables cont_substate1, end_substate1, cont_substate2 and end_substate2 ensure the synchronization of the concurrent composite state and its substates in a join.

Rule 14 *An output action* $\overline{gc}\langle t\ f\rangle \in \mathcal{A}_{out}$ *with* $arity(\overline{gc}\langle t\ f\rangle) = 2$ *which denotes a guard-condition* $gc \in GCond$ *is translated into a Boolean variable gc.*

$$\underline{A(\vec{x}) \stackrel{\text{def}}{=} event(x).([x = e_1](\boldsymbol{\nu} t\ f)\underline{\overline{gc}\langle t\ f\rangle}.}$$
$$(\underline{t.\overline{step}.B(\vec{x})} +$$
$$f.\overline{step}.A(\vec{x})) +$$
$$\Sigma_{i\neq 1}[x = e_i]\overline{step}.A(\vec{x}))$$

```
next(state) :=
case
  state = A & event_buff = e_1 & gc & !step : B;
esac;
next(step) :=
case
  state = A & event_buff = e_1 & gc & !step : 1;
esac;
```

Rule 15 *An initial configuration of a statechart diagram, which is specified as an event processor configuration in the π-calculus, determines the values of the init statements for variables state, substate, substate*1 *and substate*2 *in Rules 6, 10, 11, 12 and 13.*

Rule 14 specifies that an output action representing a guard-condition is modelled as a Boolean variable. Rule 15 stipulates that the init statements for variables which keep track of the current active state and substates are derived from the initial configuration.

Rule 16 *The consumption and retaining of an event represented as a channel are encoded as:*

```
next(event_buff) :=
case
```

```
    event_buff !=empty & next(step)      : empty;
    1                                    : event_buff;
  esac;
```

The assigning of value *empty* to *event_buff* signifies the consumption of an event. The retaining of an event is mapped to the default case of the next statement of *event_buff*.

Rule 17 *Given a statechart diagram $F \in \mathcal{SC}$ with send actions represented in the π-calculus as $\overline{ins_{G_i}}\langle e_i \rangle$ for $i = 1, \ldots, n$ which insert event E_i modelled as e_i to the event queue of statechart diagram $G_i \in \mathcal{SC}$. The parameters of module F is defined as:*

```
MODULE F(event_buff, step, ins_G_1, G_1_q_buff, ..., ins_G_n,
         G_n_q_buff)
```

Each module representing a statechart diagram contains parameters *event_buff* and *step*. The other parameters of the module are based on the send actions as defined in Rule 10.

5.4 Correctness of the Translation

Theorem 13 *There is a semantic correspondence between the firing of a transition represented in UML statecharts based sublanguage of the π-calculus and its implementation in the NuSMV language.*

Proof sketch. We consider the following cases:

Case 1. Let $F, G \in \mathcal{SC}, S_1, S_2 \in States(F)$ and a transition which consists of $E_1 \in \mathcal{E}, g_1 \in GCond$ and $act_1 \in Act$ defined as send $obj_G.E_2$ connecting S_1 to S_2 such that G is an associated statechart diagram of obj_G. Suppose $S_1, S_2 \in ST_{NCS}$. Since $\phi_{state}(S_1) = S_1(step, event_S, \vec{e}, g_1, ins_G)$, $\phi_{event}(E_1) = e_1$, $\phi_{guard}(g_1) = \overline{g_1}\langle t\ f \rangle$, $\phi_{action}($ send $obj_G.E_2) = \overline{ins_G}\langle e_2 \rangle$ and $\phi_{state}(S_2) = S_2(step, event_S, \vec{e}, g_1, ins_G)$, we get a sequence of reductions specified as $S_1(step, event_S, \vec{e}, g_1, ins_G) \xrightarrow{event_S(x)} \xrightarrow{[x=e_1]\overline{g_1}\langle t\ f \rangle} \xrightarrow{t} \xrightarrow{\overline{ins_G}\langle e_2 \rangle} \xrightarrow{\overline{step}} S_2(step, event_S, \vec{e}, g_1, ins_G)$ which corresponds to conditions state = S_1 & event_buff= e_1 & g_1 that require to hold true before setting the values of ins_G to 1, G_q_buff to e_2, step to 1 and state to S_2.

Case 2. Let $F \in \mathcal{SC}, S_1, S_2, V_1 \in States(F)$ and a transition which consists of $E_1 \in \mathcal{E}$ connecting S_1 to S_2. Suppose $S_1 \in ST_{NCCS}$, $S_2, V_1 \in ST_{NCS}$ and $V_1 \in substates(S_1)$.

Since $\phi_{state}(S_1) = S_1(step, event_S, \vec{e}, event_V, pos, neg)$, $\phi_{state}(V_1) = V_1(event_V, \vec{e}, pos, neg)$, $\phi_{event}(E_1) = e_1$, $\phi_{state}(S_2) = S_2(step, event_S, \vec{e}, pos, neg)$ *and* $\phi^{-1}_{state}(V_1(event_V, \vec{e},\ pos,\ neg)) \in substates(\phi^{-1}_{state}(S_1(step, event_S, \vec{e}, event_V, pos, neg)))$, *we get two sequences of reductions.*

The first sequence of reductions $S_1(step, event_S, \vec{e}, event_V, pos, neg) \xrightarrow{event_S(x)} \xrightarrow{\overline{event_V}\langle x\ ack\rangle} \xrightarrow{ack(y)} \xrightarrow{[y=pos]\overline{step}} S_2(step, event_S, \vec{e}, pos, neg)$ *implies* $S_1(step, event_S, \vec{e},\ event_V, pos, neg) \xrightarrow{event_S(x)} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{[y=pos]\overline{step}} S_2(step, event_S, \vec{e}, pos, neg)$ *as* $\overline{event_V}\langle x\ ack\rangle$ *and* $ack(y)$ *are unobservable actions. This corresponds to conditions* $state = S\_1$ & $ack = pos$ *that require to hold true before setting the values of step and state to* 1 *and* S_2.

The second sequence of reductions $V_1(event_V, \vec{e}, pos, neg) \xrightarrow{event_V(x\ ack)} \xrightarrow{[x=e_1]\overline{ack}\langle pos\rangle} \mathbf{0}$ *corresponds to conditions* $substate = V\_1$ & $event\_buff = e\_1$ *that require to hold true before setting the values of ack and substate are set to pos and nil.*

Case 3. Let $F \in \mathcal{SC}, S_2, V_2, W_2, S_3 \in States(F)$, *two transitions connect* V_2 *and* W_2, *respectively, to a join pseudostate and a transition which consists of* $E_1 \in \mathcal{E}$ *connecting the join pseudostate to* S_3. *Suppose* $S_2 \in ST_{CCS}$, $V_2, W_2, S_3 \in ST_{NCS}$ *and* V_2, W_2 *are substates of* S_2 *which are located in two different orthogonal regions. Since* $\phi_{state}(S_2) = S_2(step, event_S, \vec{e}, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn})$, $\phi_{state}(V_2) = V_2(event_{substate1}, \vec{e}, \widetilde{ack}, cont_{substate1}, end_{substate1})$, $\phi_{state}(W_2) = W_2(event_{substate2}, \vec{e}, \widetilde{ack},\ cont_{substate2}, end_{substate2})$, $\phi_{event}(E_1) = e_1$, $\phi_{state}(S_3) = S_3(step, event_S, \vec{e}, \widetilde{ack})$ *such that* $\widetilde{ack}$ *abbreviates* pos, neg *and* $\widetilde{syn}$ *abbreviates* $cont_{substate1}, end_{substate1}, cont_{substate2}, end_{substate2}$, *we get three sequences of reductions.*

The first sequence of reductions $S_2(step, event_S, \vec{e},\ event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn}) \xrightarrow{event_S(x_1)} \xrightarrow{\overline{event_{substate1}}\langle x_1\ ack_{substate1}\rangle} \xrightarrow{\overline{event_{substate2}}\langle x_1\ ack_{substate2}\rangle} \xrightarrow{ack_{substate1}(x_2)} \xrightarrow{ack_{substate2}(x_3)} \xrightarrow{[x_2=pos][x_3=pos]\overline{end_{substate1}}} \xrightarrow{\overline{end_{substate2}}} \xrightarrow{\overline{step}} S_3(step, event_S, \vec{e}, \widetilde{ack})$ *implies* $S_2(step, event_S,\ \vec{e}, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn}) \xrightarrow{event_S(x_1)} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{\tau} \xrightarrow{[x_2=pos][x_3=pos]\overline{end_{substate1}}} \xrightarrow{\overline{end_{substate2}}} \xrightarrow{\overline{step}} S_3(step, event_S, \vec{e}, \widetilde{ack})$ *as* $\overline{event_{substate1}}\langle x_1\ ack_{substate1}\rangle$, $\overline{event_{substate2}}\langle x_1\ ack_{substate2}\rangle$, $ack_{substate1}(x_2)$ *and* $ack_{substate2}(x_3)$ *are unobservable actions. This corresponds to conditions* $state = S\_2$ & $end\_substate1$ & $end\_substate2$ *require to hold true before setting the values of step to* 1 *and state to* S_3.

The second sequence of reductions $V_2(event_{substate1}, \vec{e}, \widetilde{ack}, cont_{substate1}, end_{substate1}) \xrightarrow{event_{substate1}(x\ ack_{substate1})} \xrightarrow{[x=e_3]\overline{ack_{substate1}}\langle pos\rangle} \xrightarrow{end_{substate1}} \mathbf{0}$ *corresponds to (i) condi-*

*tions substate*1 = V_2 & *event_buff* = e_3 & *end_substate*1 *require to hold true before setting the value of substate*1 *to nil; and (ii) conditions substate*1 = V_2 & *event_buff* = e_3 *require to hold true before setting the value of ack_substate*1 *to pos.*

The argument for the third sequence of reductions is similar except V_2 *is replaced by* W_2, $event_{substate1}$ *is replaced by* $event_{substate2}$, $cont_{substate1}$ *is replaced by* $cont_{substate2}$, $end_{substate1}$ *is replaced by* $end_{substate2}$, $ack_{substate1}$ *is replaced by* $ack_{substate2}$, *substate*1 *is replaced by substate*2, V_2 *is replaced by* W_2 *and end_substate*1 *is replaced by end_substate*2 *and ack_substate*1 *is replaced by ack_substate*2. *Since the converse holds true by analogous arguments and similar arguments hold for other cases, this completes the proof the statement.*

Theorem 14 *There is a semantic correspondence between UML statecharts based sublanguage of the* π*-calculus and its implementations in the NuSMV language.*
Proof sketch. Analogous to Theorem 2.

5.5 Related Work

Representative works on the analysis of UML statechart diagrams using model checking include Latella et al. [60, 38, 39] and Lilius and Paltor [62]. UML statechart diagrams are translated into PROMELA, which is the input language of SPIN, using EHA as an intermediate representation in [60]. Likewise, UML statechart diagrams are transformed into JACK environment using EHA as an intermediate representation in [38, 39]. Unlike the works of Latella et al. [60, 38, 39] which use EHA as an intermediate representation, UML statechart diagrams are translated into PROMELA using rewrite rules as an intermediate representation in the work of Lilius and Paltor [62]. Other related work which focuses on statecharts rather than on UML statechart diagrams include Mikk et al. [72]. The approach taken by Mikk et al. [72] is the same as one of the studies [60] of Latella et al. in which statecharts are first transformed into EHA and then into PROMELA. When compared with these previous studies, our approach has advantages over them in two aspects. Firstly, the π-calculus which is an intermediate representation of our approach is a formal method. It supports the equivalence checking of statechart diagrams which other intermediate representations cannot deal with. Secondly, Latella et al. [60], Lilius and Paltor [62] and Mikk et al. [72] adopt SPIN [48] as the model checker. The SPIN model checker supports linear temporal logic. In the

two other studies [38, 39] of Latella et al., the JACK environment [15] that is based on branching temporal logic is selected as the model checker. When compared with SPIN and JACK, the advantage of the NuSMV model checker is it has a higher expressibility as it supports both linear and branching temporal logics.

5.6 Summary

This chapter has presented a new methodology for analyzing finite state systems. It models a system as statechart diagrams, translates the statechart diagrams and their execution semantics into the π-calculus and encodes the π-calculus expressions in the NuSMV input language. A demonstration on the use of the proposed approach for verifying the non-security aspects of the SET/A protocol is given in the next chapter.

Chapter 6

Evaluation of the Integrated Approach

To facilitate the exchange of goods and services over the Internet, researchers have proposed a number of payment protocols [3]. Some of the more well-known payment protocols include Digicash [24], iKP [5], MicroMint [99], Millicent [80, 81], Mondex [80, 81, 95], NetBill [32], NetCheque [80, 81], PayWord [99] and Secure Electronic Transaction (SET) [70].

This chapter aims at illustrating the application of the integrated approach proposed in Chapter 5 for analyzing an agent-based payment protocol and a statechart diagram consisting of a concurrent composite state. The SET/A protocol [100], which is an agent-based payment protocol based on the SET protocol [70], is used as a running example throughout the rest of this thesis.

Secure Electronic Transaction (SET) [70] is a payment protocol which was developed by the two credit card companies Visa and MasterCard. It provides secure credit card payment over public networks such as the Internet. SET/A [100] is an agent-based payment protocol which is based on the SET protocol. It is designed for secure credit card payment in a mobile computing environment. The agent used in the protocol is a mobile agent.

A mobile agent is a software agent [16] which travels from one computer to another remote computer. Unlike a remote procedure call (RPC) which a process interacts with a remote process over a network, a mobile agent interacts with a remote process on the same computer by travelling over there. The application of mobile agents in payment

protocols poses some new challenges:

1. A payment protocol often involves various parties which are distributed over a network. Unlike traditional payment protocols, in an agent-based payment protocol a customer's agent travels to a merchant's server for carrying out a transaction. How can we formally specify and verify the correctness of this new type of agent-based payment protocol?
2. Can the various parties which are involved in an agent-based protocol reach a consensus if an agent fails while it is travelling to a merchant's server over an unreliable transmission medium?
3. As there is no way to guarantee that a mobile agent that moves to a merchant's server does not have a run-time error, can non-faulty parties agree on a decision if the agent crashes after arriving at the merchant's server?

An integrated approach which directly addresses the first question has been proposed in Chapter 5. The design of a protocol is first specified in UML statechart diagrams, then formalized in the π-calculus and finally verified automatically using NuSMV. Through model-based analyses, the validity of the second and third questions for a particular agent-based payment protocol can also be determined. In particular, we illustrate an application of the integrated approach for verifying the SET/A protocol via model checking in this chapter. The main goals of the verification are to show that the cardholder and payment gateway have reached the same decision and the protocol is deadlock-free.

The reminder of this chapter is structured as follows. Section 6.1 discusses related work in the area. The SET/A protocol is given in Section 6.2. Section 6.3 describes how the SET/A protocol is represented in UML statechart diagrams. Section 6.4 demonstrates the encoding of the statechart diagrams in the π-calculus. Section 6.5 examines the implementation of the SET/A protocol in the NuSMV language. The verification of the SET/A protocol is covered in Section 6.6. Section 6.7 presents a failure analysis and extends the original SET/A protocol to tolerate an agent failure. The evaluation of the proposed approach by a statechart diagram which contains a concurrent composite state is provided in Section 6.8. Section 6.9 concludes the chapter.

This chapter is an extended version of the paper that appeared in [57].

6.1 Related Work

Odell et al. [79, 4] extend the UML for specifying agent systems. However, the proposed Agent UML (AUML) addresses only part of the first question as it does not support the verification of agent systems.

Previous studies on the verification of e-commerce protocols using model checking include Heintze et al. [46], Lowe and Roscoe [63], Lu and Smolka [65], Ray and Ray [97], etc. All these 4 studies first encode an e-commerce protocol in Communicating Sequential Processes (CSP) [47] and then formally analyze the model using the Failures-Divergence Refinement (FDR) model checker [64].

Our approach is a combination of UML statechart diagrams and the NuSMV model checker in which the π-calculus is used as an internal representation. In contrast to [79, 4], our approach allows the analysis and verification of protocols using NuSMV. In addition, it has an advantage over [46, 63, 65, 97] by providing a systematic way for checking whether the alternative representations of a protocol in statechart diagrams are equivalent or not through the use of the π-calculus.

6.2 The SET/A Protocol

The SET/A protocol focuses on the initiation, purchase and authorization phases. The steps of the protocol are given in Figure 6.1.

Step 1: The cardholder C sends a dispatch request which contains the purchase request $PchaseReq$, its signature certificate $Cert_C$, the order information OI and the payment instructions PI to the agent A.

Step 2: Upon arrival at the merchant's server, the agent A sends an initialization request to the merchant M.

Step 3: M sends a response which includes the unique transaction identifier $TransId$, its signature certificate $Cert_M$ and the payment gateway's key-exchange certificate $Cert_{PG,X}$.

Step 4: The agent A computes the hash values $H[PI]$ and $H[OI]$ for PI and OI, respectively. A dual signature DS is generated by using the $H[PI]$ and $H[OI]$ pair. To prevent M from obtaining the PI, a randomly generated symmetric key K is used for encrypting PI, DS and $H[OI]$. A digital envelope which contains the key K is then created by encrypting K with the payment gateway's

1. $C \rightarrow A$: dispatch request
   $\langle PchaseReq, Cert_C, OI, PI \rangle$
2. $A \rightarrow M$: initialization request
   $\langle InitReq \rangle$
3. $M \rightarrow A$: initialization response
   $\langle TransId, Cert_M, Cert_{PG,X} \rangle$
4. $A \rightarrow M$: purchase request
   $\langle TransId, Cert_C, OI, DS, H[PI],\ E_{PG,X-Pub}[K], E_K[PI, DS, H[OI]] \rangle$
5. $M \rightarrow P$: authorization request
   $\langle TransId, E_{PG,X-Pub}[K], E_K[PI, DS, H[OI]] \rangle$
6. $P \rightarrow M$: authorization response
   $\langle TransId, AuthCode \rangle$
7. $M \rightarrow A$: purchase response
   $\langle TransId, AuthCode, Cert_M \rangle$
8. $A \rightarrow C$: dispatch response
   $\langle AuthCode \rangle$

Figure 6.1: The SET/A Protocol

public key-exchange key. Eventually, A sends all these with the transaction identifier, the cardholder's certificate and the OI to M.

Step 5: M checks the validity of OI using DS and $H[PI]$. Then M forwards the digital envelope, the encrypted PI, the encrypted DS and the encrypted $H[OI]$ to the payment gateway P for payment authorization.

Step 6: Upon receipt of an authorization request, P verifies the correctness of PI and sends an authorization response which includes the transaction identifier $TransId$ and the authorization code $AuthCode$ to M.

Step 7: After receiving an authorization response, M forwards the response with its signature certificate to A.

Step 8: A departs the merchant's server, arrives at the cardholder's computer and sends a response to C.

6.3 Modelling the SET/A Protocol Using Statechart Diagrams

As mentioned in preceding chapters, the statechart diagrams of UML are a graphical notation for specifying and visualizing the dynamic aspects of a system. The modelling of the SET/A protocol using UML statechart diagrams focuses on the messages exchanged among the cardholder, agent, merchant and payment gateway. It abstracts away the low level cryptographic mechanisms as the desired properties to be verified are at a higher level.

Figures 6.2 and 6.3 depict the UML statechart diagrams for the cardholder and agent. A transaction begins when the cardholder browses the merchant's catalog, submits a purchase request (*cPchaseReq*) and sends a dispatch request (*cDpatchReq*) to the agent. The state *CBrowseCatalog* is exited and the state *CWaitPchaseRespn* is entered. The cardholder continues to wait in state *CWaitPchaseRespn* until either a positive or negative dispatch response is received. The transaction terminates when the cardholder enters a commit or an abort state.

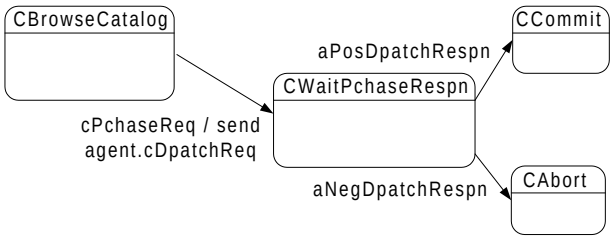

Figure 6.2: Statechart diagram of the cardholder

On receiving the dispatch request (*cDpatchReq*), the agent which is in an idle state (*AIdle*) travels from the cardholder's computer to the merchant's server. Upon arrival at the merchant's server (*aArvdSrvr*), the agent sends an initialization request (*aInitReq*) to the merchant. The agent waits for an initialization response (*mInitRespn*), generates a purchase request (*aPchaseReq*) to the merchant and blocks until a positive or negative purchase response is received. The agent then departs the merchant's server, travels back to the cardholder's computer and sends a positive or negative dispatch response to the cardholder.

The statechart diagrams for the merchant and payment gateway are shown in Fig-

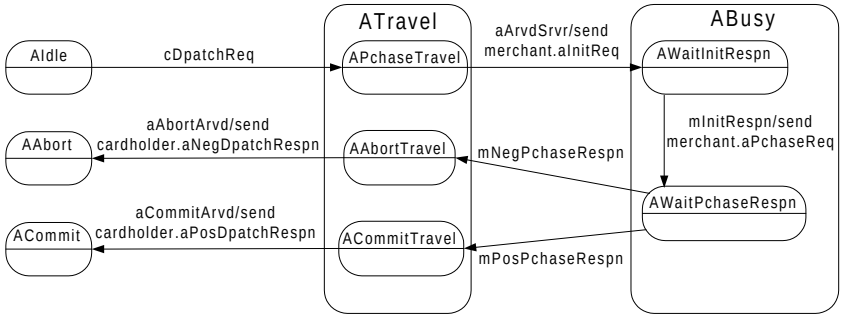

Figure 6.3: Statechart diagram of the agent

ures 6.4 and 6.5. Upon receipt of the initialization request (*aInitReq*) from the agent, the merchant returns an initialization response (*mInitRespn*) to the agent, waits for a purchase request (*aPchaseReq*) and sends an authorization request (*mAuthReq*) to the payment gateway. Depending on whether a positive or negative authorization response is received, a positive or negative purchase response is returned to the agent.

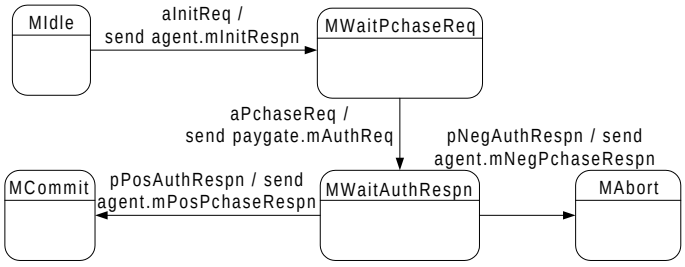

Figure 6.4: Statechart diagram of the merchant

Based on the issuer's approval code, the payment gateway responds to the authorization request (*mAuthReq*) by sending a positive or negative authorization response to the merchant.

6.4 Encoding the Statechart Diagrams in the π-Calculus

In this section, we present a π-calculus representation for the statechart diagram of the cardholder. This is then followed by a discussion on how an interlevel transition in the

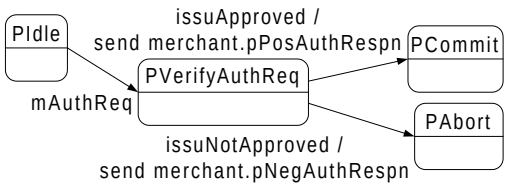

Figure 6.5: Statechart diagram of the payment gateway

statechart diagram of the agent is modelled in the π-calculus.

To make the π-calculus specifications easier to read, we define an abbreviation as follows:

$$\widetilde{e_C} = cPchaseReq, cDpatchReq, aPosDpatchRespn, aNegDpatchRespn$$

The four events in Figure 6.2 are mapped to four channels *cPchaseReq*, *cDpatchReq*, *aPosDpatchRespn* and *aNegDpatchRespn* in the π-calculus according to Rule 1 of Section 3.2. Applying Rule 2 of Section 3.2, the state *CBrowseCatalog* is represented in the π-calculus by the following pattern:

$$\begin{aligned} &CBrowseCatalog(step, event_C, \widetilde{e_C}, ins_{agent}) \stackrel{\text{def}}{=} \\ &\quad event_C(x). \\ &\quad ([x = cPchaseReq] \ldots + \\ &\quad \Sigma_{\mathrm{e}\in\{\widetilde{\mathrm{e}_C}\}\setminus\{cPchaseReq\}}[x = e] \ldots) \end{aligned}$$

The send action (Definition 14):

$$send\ agent.cDpatchReq$$

based on Rule 4 of Section 3.2 is denoted by:

$$\overline{ins_{agent}}\langle cDpatchReq\rangle$$

The end of a run-to-completion step is expressed as $\overline{step}$. The complete π-calculus

specification of the state *CBrowseCatalog* in Figure 6.2 is then described by:

$$
\begin{aligned}
&CBrowseCatalog(step, event_C, \widetilde{e_C}, ins_{agent}) \stackrel{\text{def}}{=} \\
&\quad event_C(x). \\
&\quad ([x = cPchaseReq]\overline{ins_{agent}}\langle cDpatchReq\rangle.\overline{step}. \\
&\quad\ CWaitPchaseRespn(step, event_C, \widetilde{e_C}, ins_{agent}) + \\
&\quad\ \Sigma_{\mathrm{e}\in\{\widetilde{\mathrm{e}_C}\}\setminus\{cPchaseReq\}}[x = e]\overline{step}. \\
&\quad\ CBrowseCatalog(step, event_C, \widetilde{e_C}, ins_{agent}))
\end{aligned}
$$

Likewise, the behaviour of the state *CWaitPchaseRespn* in which the two transitions are in conflict (Definition 9 and Subsection 3.4.2) is specified by:

$$
\begin{aligned}
&CWaitPchaseRespn(step, event_C, \widetilde{e_C}, ins_{agent}) \stackrel{\text{def}}{=} \\
&\quad event_C(x). \\
&\quad ([x = aPosDpatchRespn]\overline{step}.CCommit(step, event_C, \widetilde{e_C}, ins_{agent}) + \\
&\quad\ [x = aNegDpatchRespn]\overline{step}.CAbort(step, event_C, \widetilde{e_C}, ins_{agent}) + \\
&\quad\ \Sigma_{\mathrm{e}\in\{\widetilde{\mathrm{e}_C}\}\setminus\{aPosDpatchRespn, aNegDpatchRespn\}}[x = e]\overline{step}. \\
&\quad\ CWaitPchaseRespn(step, event_C, \widetilde{e_C}, ins_{agent}))
\end{aligned}
$$

We now illustrate how an interlevel transition which connects the substate *APchaseTravel* to the substate *AWaitInitRespn* (see Figure 6.3) is encoded in the π-calculus. We assume that *ATravel* and *APchaseTravel* are the active states of the statechart diagram. In addition, the following abbreviations are defined to improve the readability of the specifications:

$$
\begin{aligned}
&\widetilde{e_A} = cDpatchReq, aArvdSrvr, aInitReq, \\
&\qquad mInitRespn, aPchaseReq, mNegPchaseRespn, mPosPchaseRespn, \\
&\qquad aAbortArvd, aCommitArvd, aNegDpatchRespn, aPosDpatchRespn \\
&\widetilde{ins_A} = ins_{cardholder}, ins_{merchant} \\
&\widetilde{ack} = pos, neg
\end{aligned}
$$

The non-concurrent composite state *ATravel* broadcasts any received events to its active substate *APchaseTravel* along channel $event_{sub1}$. It then blocks until a signal

is received along channel *ack* (Rule 5 of Section 3.2) as specified by the following π-calculus specification:

$$
\begin{aligned}
&ATravel(step, event_A, event_{sub1}, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack}) \stackrel{\text{def}}{=} \\
&\quad event_A(x_1).(\boldsymbol{\nu} ack)\overline{event_{sub1}}\langle x_1\ ack\rangle.ack(x_2). \\
&\quad ([x_2 = pos]([x_1 = aArvdSrvr]\overline{step}.(\boldsymbol{\nu} event_{sub2}) \\
&\quad\ (ABusy(step, event_A, event_{sub2}, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack})| \\
&\quad\ AWaitInitRespn(event_{sub2}, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack})) + \\
&\quad [x_1 = aAbortArvd]\overline{step}.AAbort(step, event_A, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack}) + \\
&\quad [x_1 = aCommitArvd]\overline{step}.ACommit(step, event_A, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack})) + \\
&\quad [x_2 = neg]\overline{step}.ATravel(step, event_A, event_{sub1}, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack}))
\end{aligned}
$$

If a positive acknowledgement is received and the received channel is *aArvdSrvr*, *ATravel* sends a signal along *step* and continues as two concurrent processes *ABusy* and *AWaitInitRespn*. If a negative acknowledgement is received, *ATravel* signals the completion of the step and continues as itself. Rule 5 of Section 3.2 stipulates that the substate *APchaseTravel* is defined by:

$$
\begin{aligned}
&APchaseTravel(event_{sub1}, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack}) \stackrel{\text{def}}{=} \\
&\quad event_{sub1}(x\ ack). \\
&\quad ([x = aArvdSrvr]\overline{ins_{merchant}}\langle aInitReq\rangle.\overline{ack}\langle pos\rangle + \\
&\quad\ \Sigma_{\mathrm{e}\in\{\widetilde{\mathrm{e}_A}\}\setminus\{aArvdSrvr\}}[x = e]\overline{ack}\langle neg\rangle. \\
&\quad\ APchaseTravel(event_{sub1}, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack}))
\end{aligned}
$$

The substate *APchaseTravel* receives events along channel $event_{sub1}$ from its superstate *ATravel*. On receipt of the channel *aArvdSrvr*, it outputs the event *aInitReq*, generates the positive acknowledgement and terminates itself.

The complete SET/A protocol is translated to its π-calculus representation in a similar manner. A translator called SC2PiCal, which we shall describe in next chapter, has been developed for automating the transformation process. The formalization of statechart diagrams in the π-calculus is compositional in which a system which consists of multiple communicating statechart diagrams are translated separately into their π-calculus equivalent representations.

6.5 Implementing the SET/A Protocol in the NuSMV Language

Following the translation rules proposed in Chapter 5, we translate the 4 statechart diagrams represented in the π-calculus into the NuSMV language. Each statechart diagram is implemented in the NuSMV language as a module. The construction of the SET/A protocol in the NuSMV input language has adopted an asynchronous model. All 4 parties (modules) are executing in parallel which is based on an interleaving semantics. In NuSMV, asynchronism is modelled by instantiating all modules with the keyword *process*. One of these modules is then non-deterministically selected to execute at a time.

The statechart diagram of the cardholder is coded in the NuSMV language (Rules 1 and 17 of Chapter 5) as line 1 in Figure 6.6.

The four π-calculus process identifiers $CBrowseCatalog(step, event_C, \widetilde{e_C}, ins_{agent})$, $CWaitPchaseRespn(step, event_C, \widetilde{e_C}, ins_{agent})$, $CCommit(step, event_C, \widetilde{e_C}, ins_{agent})$ and $CAbort(step, event_C, \widetilde{e_C}, ins_{agent})$ are encoded as four symbolic values (Rule 2 of Chapter 5) as lines 2 and 3.

The channels $cPchaseReq, cDpatchReq$, $aPosDpatchRespn$ and $aNegDpatchRespn$ are implemented in NuSMV as symbolic values (Rule 5 of Chapter 5). Consider the π-calculus specification of the state $CBrowseCatalog$:

$$
\begin{aligned}
&CBrowseCatalog(step, event_C, \widetilde{e_C}, ins_{agent}) \stackrel{\text{def}}{=} \\
&\quad event_C(x). \\
&\quad ([x = cPchaseReq]\overline{ins_{agent}}\langle cDpatchReq\rangle.\overline{step}. \\
&\quad CWaitPchaseRespn(step, event_C, \widetilde{e_C}, ins_{agent}) + \\
&\quad \Sigma_{\mathrm{e}\in\{\widetilde{\mathrm{e}_C}\}\setminus\{cPchaseReq\}}[x = e]\overline{step}. \\
&\quad CBrowseCatalog(step, event_C, \widetilde{e_C}, ins_{agent}))
\end{aligned}
$$

Applying Rules 6–10 of Chapter 5, we get lines 6–9, 14–21 and 26–41 of Figures 6.6 and 6.7. The value of the variable *state* is set to *CWaitPchaseRespn* when the pre-condition on lines 8 and 9 holds. The expression assigns the value 1 (*true*) to the Boolean variable *step* for indicating the end of a run-to-completion step on lines 18 and 19. The sending of the event *cDpatchReq* to the agent is regarded as setting the values of *a_q_buff* and *ins_agent* to *cDpatchReq* and 1 (*true*) as shown on lines

```
1 MODULE cardholder(event_buff, a_q_buff, step, ins_agent)
2 VAR
3   state: {CBrowseCatalog, CWaitPchaseRespn, CCommit, CAbort};
4 ASSIGN
5   init(state) := CBrowseCatalog;
6   next(state) :=
7     case
8       state = CBrowseCatalog & event_buff = cPchaseReq &
9             !ins_agent & !step                                  : CWaitPchaseRespn;
10      state = CWaitPchaseRespn & event_buff = aPosDpatchRespn &
11            !step                                               : CCommit;
12      state = CWaitPchaseRespn & event_buff = aNegDpatchRespn &
13            !step                                               : CAbort;
14      1                                                         : state;
15    esac;
16  next(step) :=
17    case
18      state = CBrowseCatalog & event_buff = cPchaseReq &
19            !ins_agent & !step                                  : 1;
20      state = CBrowseCatalog & event_buff = cPchaseReq &
21            ins_agent & !step                                   : step;
22      state = CWaitPchaseRespn & event_buff = aPosDpatchRespn &
23            !step                                               : 1;
24      state = CWaitPchaseRespn & event_buff = aNegDpatchRespn &
25            !step                                               : 1;
26      state = next(state) & !step                               : 1;
27      event_buff != empty & step                                : 0;
28      1                                                         : step;
29    esac;
30  next(a_q_buff) :=
31    case
32      state = CBrowseCatalog & event_buff = cPchaseReq &
33            !ins_agent & !step                                  : cDpatchReq;
34      1                                                         : a_q_buff;
35    esac;
```

Figure 6.6: NuSMV source code for the cardholder (lines 1–35)

30–33 and lines 36–39. Rule 15 of Chapter 5 stipulates that the initial value of *state* is set to $CBrowseCatalog$ (line 5). According to Rule 16 of Chapter 5, we get the NuSMV code on lines 42–46. Likewise, we implement the π-calculus specification of state $CWaitPchaseRespn$ in the NuSMV code that corresponds to lines 10–13 and lines 22–25 of Figure 6.6.

In the remainder of this section, we examine how the interlevel transition, the non-

```
36  next(ins_agent) :=
37    case
38      state = CBrowseCatalog & event_buff = cPchaseReq &
39              !ins_agent & !step                              : 1;
40      1                                                       : ins_agent;
41    esac;
42  next(event_buff) :=
43    case
44      event_buff !=empty & next(step)                         : empty;
45      1                                                       : event_buff;
46    esac;
47 FAIRNESS running
```

Figure 6.7: NuSMV source code for the cardholder (lines 36–47)

concurrent composite state $ATravel$ and the substate $APchaseTravel$ are implemented in the NuSMV language.

Rule 4 of Chapter 5 states that the process identifier $APchaseTravel(event_{sub1}, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack})$ representing a substate is coded as:

```
VAR
substate: {APchaseTravel, ...};
```

Direct substates of the non-concurrent composite state *ATravel* are modelled as a scalar variable *substate*. As the state *APchaseTravel* is a substate of the non-concurrent composite state *ATravel*, the substate *APchaseTravel* is encoded in the NuSMV language as a symbolic value of the scalar variable *substate*.

Using Rule 11 of Chapter 5, the interlevel transition which is specified by the following fragments of π-calculus specifications:

$$
\begin{aligned}
&ATravel(step, event_A, event_{sub1}, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack}) \stackrel{\text{def}}{=} \\
&\quad event_A(x_1).(\boldsymbol{\nu} ack)\overline{event_{sub1}}\langle x_1\ ack\rangle.ack(x_2). \\
&\quad ([x_2 = pos]([x_1 = aArvdSrvr]\overline{step}.(\boldsymbol{\nu} event_{sub2}) \\
&\quad\ (ABusy(step, event_A, event_{sub2}, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack})| \\
&\quad\ AWaitInitRespn(event_{sub2}, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack})) + \\
&\quad\ \ldots) + \\
&\quad [x_2 = neg]\overline{step}.ATravel(step, event_A, event_{sub1}, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack}))
\end{aligned}
$$

$$
\begin{aligned}
&APchaseTravel(event_{sub1}, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack}) \stackrel{\text{def}}{=} \\
&\quad event_{sub1}(x\ ack). \\
&\quad ([x = aArvdSrvr]\overline{ins_{merchant}}\langle aInitReq\rangle.\overline{ack}\langle pos\rangle + \\
&\quad \Sigma_{\mathrm{e}\in\{\widetilde{\mathrm{e}_A}\}\setminus\{aArvdSrvr\}}[x = e]\overline{ack}\langle neg\rangle. \\
&\quad APchaseTravel(event_{sub1}, \widetilde{e_A}, \widetilde{ins_A}, \widetilde{ack}))
\end{aligned}
$$

is encoded in NuSMV as shown in Figure 6.8. Lines 20 and 21 set the value of *substate* to *nil* (Rules 3 and 11 of Chapter 5) for signifying that there is not any active substate. A positive acknowledgement is generated by assigning the value *pos* to the scalar variable *ack* (lines 25 and 26). The sending of the event *aInitReq* to the merchant is represented by setting the values of *m_q_buff* and *ins_merchant* to *aInitReq* and 1 (*true*) on lines 34, 35, 39 and 40. The Boolean variable *superstate_enable* (line 28) and scalar variable *ack* model the lower-first firing priority scheme. Lines 8, 9, 13, 14, 18 and 19 represent that the target states *ABusy* and *AWaitInitRespn* are entered. The fairness constraint on line 42 (47) in Figure 6.8 (6.7) specifies that the module is selected for execution infinitely often.

Using a similar approach, we implement the dispatcher and the event queue as a module in the NuSMV input language. Only one module is required as the functionality of the dispatcher has been incorporated into the event queue. The event queue enqueues, dequeues and dispatches one event at a time to the event processor. In addition, the implementation of the event processor is optimized by eliminating the root state so that a dispatched event is directly received by one of the substates of the root state.

6.6 Verification of the SET/A Protocol

After the construction of the model, we now verify the correctness of the SET/A protocol. Our verification focuses on data integrity and deadlock freedom properties. Data integrity property states that the response which is received by the cardholder is consistent with the response which is sent out by the payment gateway. Deadlock freedom property ensures that the sequence of messages exchanged among the four parties is free from deadlock. The complication in analyzing the sequence of messages exchanged arises from the concurrent execution of the 4 statechart diagrams (parties).

We assume that the communication channels over which the payment protocol

```
1  MODULE agent(...)
2  VAR
3    substate: {APchaseTravel, ...};
4    ...
5  ASSIGN
6   next(state) :=
7    case
8     state = ATravel & ack = pos & event_buff =
9     aArvdSrvr & !step & superstate_enable         : ABusy;
10     ...
11  next(step) :=
12   case
13    state = ATravel & ack = pos & event_buff =
14    aArvdSrvr & !step & superstate_enable         : 1;
15    ...
16  next(substate) :=
17   case
18    state = ATravel & ack = pos & event_buff =
19    aArvdSrvr & !step & superstate_enable         : AWaitInitRespn;
20    substate = APchaseTravel & event_buff =
21    aArvdSrvr & !ins_merchant & !superstate_enable: nil;
22    ...
23  next(ack) :=
24   case
25    substate = APchaseTravel & event_buff =
26    aArvdSrvr & !ins_merchant & !superstate_enable: pos;
27    ...
28  next(superstate_enable) :=
29   case
30    (next(ack)=pos | next(ack)=neg) & !step       : 1;
31    ...
32  next(m_q_buff) :=
33   case
34    substate = APchaseTravel & event_buff =
35    aArvdSrvr & !ins_merchant                     : aInitReq;
36    ...
37  next(ins_merchant) :=
38   case
39    substate = APchaseTravel & event_buff =
40    aArvdSrvr & !ins_merchant                     : 1;
41    ...
42 FAIRNESS running
```

Figure 6.8: NuSMV source code for the agent

operates are reliable. Only one instance of the payment protocol which consists of one cardholder, one agent, one merchant and one payment gateway is considered in our analysis. As an illustration, we consider the following CTL specifications:

$$\mathbf{AG}(p.state = \text{PCommit} \Rightarrow \mathbf{AF}c.state = \text{CCommit})$$

$$\mathbf{AG!}((p.state = \text{PCommit} \ \& \ c.state = \text{CAbort}) \mid (p.state = \text{PAbort} \ \& \ c.state = \text{CCommit}))$$

The first CTL formula states that if the payment gateway commits the transaction, the cardholder will also eventually commit the transaction. The second CTL formula asserts that the payment gateway and cardholder either both commit or abort the transaction.

Another desired property is to ensure that the sequence of messages exchanged among the 4 parties does not lead to a deadlock. The corresponding CTL formula is given below:

$$\mathbf{AG} \ \mathbf{AF}(c.state = \text{CCommit} \mid c.state = \text{CAbort})$$

Literally, the formula says it is always possible for the cardholder to receive a commit or an abort message.

The verification of these formulas took approximately 4 seconds on a Pentium 4 2.4GHz PC with 512MB of memory using NuSMV 2.1.2 (with -dynamic and -f options) running under the Windows XP Professional operating system.

6.7 Failure Analysis of the Protocol

In this section, we illustrate that the deadlock freedom property and one of the data integrity properties are violated in the presence of an agent failure. An extension to the original protocol is proposed for ensuring that a transaction is resilient to the failure of the mobile agent.

To analyze whether the SET/A protocol is resilient to an agent failure, we model the occurrence of a failure by adding a failure state to the original agent's statechart diagram as shown in Figure 6.9.

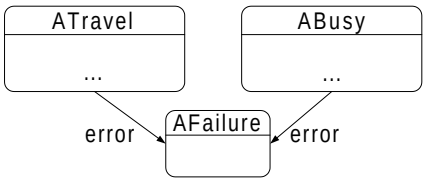

Figure 6.9: Failure state

We check the new model using NuSMV, the first data integrity property

$$\mathbf{AG}(p.state = \text{PCommit} \Rightarrow \mathbf{AF}c.state = \text{CCommit})$$

is violated in the presence of an agent failure. The property fails as there is no way for the cardholder to find out what the decision of the payment gateway is when the agent has crashed. The deadlock freedom property is also violated as the cardholder blocks whenever the agent has crashed.

To allow an agent failure, we extend the SET/A protocol with a cardholder inquiry as shown in Figures 6.10, 6.11 and 6.12. The statechart diagrams of the cardholder, agent and merchant are modified, whereas the statechart diagram of the payment gateway remains unchanged.

When a timeout occurs, the cardholder (Figure 6.10) sends an inquiry request (*cInqReq*) to the merchant and waits for an inquiry response. The transaction is committed or aborted depending on whether a positive or negative inquiry response is received.

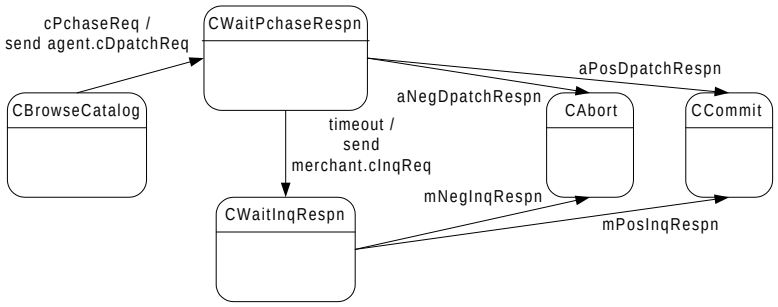

Figure 6.10: Modified statechart diagram of the cardholder

A timeout event is generated when an error (Figure 6.11) occurs. After entering the failure state, a timeout event is also generated on receipt of the event *mInitRespn*.

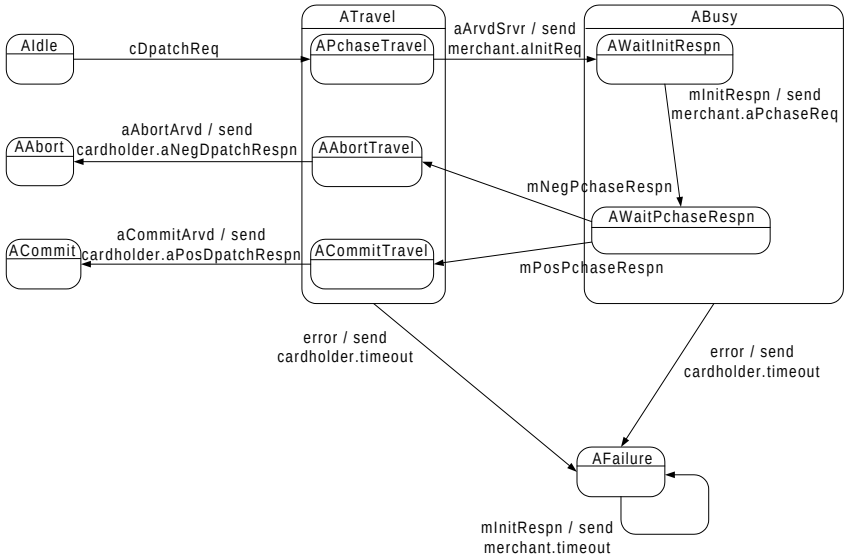

Figure 6.11: Modified statechart diagram of the agent

If the merchant receives an inquiry request (*cInqReq*) from the customer before an authorization request (*mAuthReq*) is sent to the payment gateway (i.e. before entering the state *MWaitAuthRespn* or Step 5 of the Protocol), the merchant aborts the transaction and sends a negative inquiry response (*mNegInqRespn*) to the customer. This is based on the assumption that the absence of an initialization request (*aInitReq*), a purchase request (*aPchaseReq*) or both may indicate that the agent has crashed. Otherwise, the merchant cannot abort the transaction. It requires to wait until an authorization response (*pPosAuthRespn or pNegAuthRespn*) is received from the payment gateway and sends the corresponding inquiry response (*mPosInqRespn or mNegInqRespn*) directly to the cardholder.

We encoded the revised statechart diagrams in the π-calculus and implemented them in the NuSMV language. Then we checked the data integrity properties and deadlock freedom property again using NuSMV. The extended SET/A model satisfies the desired properties and the verification took about 23 seconds.

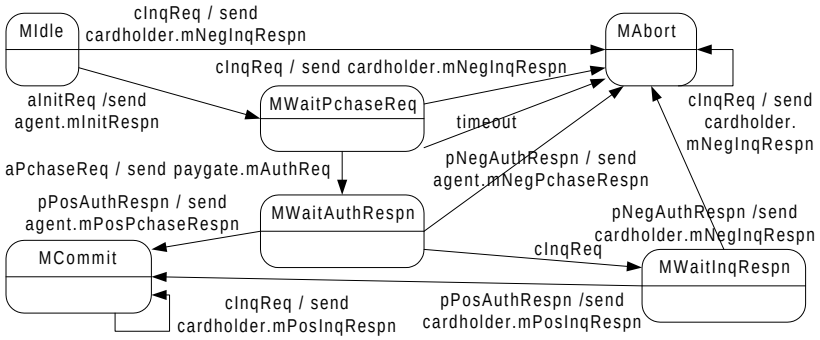

Figure 6.12: Modified statechart diagram of the merchant

6.8 Encoding and Verifying Concurrent Composite States

Although the SET/A protocol shows the application of the approach to a real-world problem, the modelling only uses the basic facilities of UML statechart diagrams. Therefore, in order to demonstrate the applicability of our approach to a broader class of problems, we analyse a small example involving a concurrent composite state.

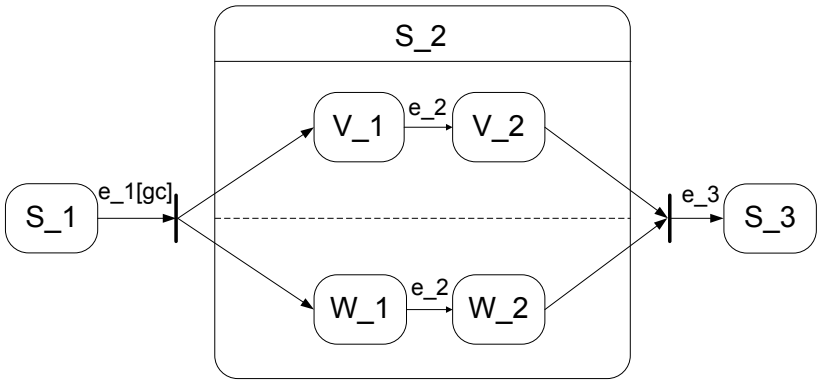

Figure 6.13: A concurrent composite state

To improve readability, the following abbreviations are defined:

$$\widetilde{e} = e\_1, e\_2, e\_3$$

$$\widetilde{ack} = pos, neg$$

$$\widetilde{syn} = cont_{substate1}, end_{substate1}, cont_{substate2}, end_{substate2}$$

The process S_1 representing state S_1 evolves to three concurrent processes S_2, V_1

and W_1 according to Rule 6 of Chapter 3 and its π-calculus specification is represented as:

$$
\begin{aligned}
&S\_1(step, event_S, \widetilde{e}, gc, \widetilde{ack}) \stackrel{\text{def}}{=} \\
&\quad event_S(x). \\
&\quad ([x = e\_1](\boldsymbol{\nu} t\ f)\overline{gc}\langle t\ f\rangle. \\
&\quad (t.\overline{step}. \\
&\quad (\boldsymbol{\nu} event_{substate1}\ event_{substate2}\ cont_{substate1}\ end_{substate1}\ cont_{substate2}\ end_{substate2}) \\
&\quad (S\_2(step, event_S, \widetilde{e}, gc, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn}) \mid \\
&\quad V\_1(event_{substate1}, \widetilde{e}, \widetilde{ack}, cont_{substate1}, end_{substate1}) \mid \\
&\quad W\_1(event_{substate2}, \widetilde{e}, \widetilde{ack}, cont_{substate2}, end_{substate2})) + \\
&\quad f.\overline{step}.S\_1(step, event_S, \widetilde{e}, gc, \widetilde{ack})) + \\
&\quad \Sigma_{e \in \{\widetilde{e}\} \setminus \{e\_1\}}[x = e]\overline{step}.S\_1(step, event_S, \widetilde{e}, gc, \widetilde{ack}))
\end{aligned}
$$

The concurrent composite state S_2 and its substates V_1 and V_2 are defined in the π-calculus as:

$$
\begin{aligned}
&S\_2(step, event_S, \widetilde{e}, gc, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn}) \stackrel{\text{def}}{=} \\
&\quad event_S(x_1).(\boldsymbol{\nu} ack_{substate1}\ ack_{substate2})\overline{event_{substate1}}\langle x_1\ ack_{substate1}\rangle. \\
&\quad \overline{event_{substate2}}\langle x_1\ ack_{substate2}\rangle.ack_{substate1}(x_2).ack_{substate2}(x_3). \\
&\quad ([x_2 = pos][x_3 = pos]\overline{end_{substate1}}.\overline{end_{substate2}}. \\
&\quad \overline{step}.S\_3(step, event_S, \widetilde{e}, gc, \widetilde{ack}) + \\
&\quad [x_2 = neg][x_3 = neg]\overline{cont_{substate1}}.\overline{cont_{substate2}}. \\
&\quad \overline{step}.S\_2(step, event_S, \widetilde{e}, gc, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn}) + \\
&\quad [x_2 = pos][x_3 = neg]\overline{cont_{substate1}}.\overline{cont_{substate2}}. \\
&\quad \overline{step}.S\_2(step, event_S, \widetilde{e}, gc, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn}) + \\
&\quad [x_2 = neg][x_3 = pos]\overline{cont_{substate1}}.\overline{cont_{substate2}}. \\
&\quad \overline{step}.S\_2(step, event_S, \widetilde{e}, gc, event_{substate1}, event_{substate2}, \widetilde{ack}, \widetilde{syn}))
\end{aligned}
$$

$$
\begin{aligned}
&V\_1(event_{substate1}, \widetilde{e}, \widetilde{ack}, cont_{substate1}, end_{substate1}) \stackrel{\text{def}}{=} \\
&\quad event_{substate1}(x\ ack_{substate1}). \\
&\quad ([x = e\_2]\overline{ack_{substate1}}\langle neg\rangle.cont_{substate1}. \\
&\quad\ V\_2(event_{substate1}, \widetilde{e}, \widetilde{ack}, cont_{substate1}, end_{substate1}) + \\
&\quad\ \Sigma_{e\in\{\widetilde{e}\}\setminus\{e\_2\}}[x = e]\overline{ack_{substate1}}\langle neg\rangle.cont_{substate1}. \\
&\quad\ V\_1(event_{substate1}, \widetilde{e}, \widetilde{ack}, cont_{substate1}, end_{substate1}))
\end{aligned}
$$

$$
\begin{aligned}
&V\_2(event_{substate1}, \widetilde{e}, \widetilde{ack}, cont_{substate1}, end_{substate1}) \stackrel{\text{def}}{=} \\
&\quad event_{substate1}(x\ ack_{substate1}). \\
&\quad ([x = e\_3]\overline{ack_{substate1}}\langle pos\rangle. \\
&\quad\ (end_{substate1} + \\
&\quad\ cont_{substate1}.V\_2(event_{substate1}, \widetilde{e}, \widetilde{ack}, cont_{substate1}, end_{substate1})) + \\
&\quad\ \Sigma_{e\in\{\widetilde{e}\}\setminus\{e\_3\}}[x = e]\overline{ack_{substate1}}\langle neg\rangle.cont_{substate1}. \\
&\quad\ V\_2(event_{substate1}, \widetilde{e}, \widetilde{ack}, cont_{substate1}, end_{substate1}))
\end{aligned}
$$

The π-calculus specifications of W_1 and W_2 are identical to the ones of V_1 and V_2 with the exception of $V\_1, event_{substate1}, cont_{substate1}, end_{substate1}, ack_{substate1}$ and V_2 are replaced by $W\_1, event_{substate2}, cont_{substate2}, end_{substate2}, ack_{substate2}$ and W_2, respectively.

Based on the translation rules defined in Chapter 5, we translate processes S_1, S_2, V_1 and V_2 into NuSMV. Applying Rules 12 and 14 of Chapter 5, we obtain lines 6–17, 19–21, 23–26, 29, 30, 32–35, 38–41, 43–54, 56–63 and 65–73 in Figures 6.14 and 6.15. Likewise, we get line 1 based on Rules 1 and 17, lines 27, 28, 36 and 37 based on Rule 13, lines 5 and 22 based on Rule 15 and lines 74–78 based on Rule 16.

Consider the case when the substate V_1 is active and an event e_2 is received, the substate V_1 is exited and the substate V_2 is entered. This property is expressed by the following CTL formula:

$$
\begin{aligned}
&\mathbf{AG}(concurr.substate1 = V\_1\ \&\ event\_buff = e\_2 \Rightarrow \\
&\quad \mathbf{AF} concurr.substate1 = V\_2)
\end{aligned}
$$

The temporal operator **F** is used instead of **X** as there are a number of intermediate states between V_1 and V_2 which model the run-to-completion step of UML state-

```
1  MODULE concurr(event_buff, step)
2  VAR
3    ...
4  ASSIGN
5    init(state) := S_1;
6    next(state) :=
7      case
8        state = S_1 & event_buff = e_1 & gc & !step            : S_2;
9        state = S_2 & end_substate1 & end_substate2 & !step    : S_3;
10       state = S_2 & cont_substate1 & cont_substate2 & !step  : S_2;
11       1                                                      : state;
12     esac;
13   next(step) :=
14     case
15       state = S_1 & event_buff = e_1 & gc & !step            : 1;
16       state = S_2 & end_substate1 & end_substate2 & !step    : 1;
17       state = S_2 & cont_substate1 & cont_substate2 & !step  : 1;
18       ...
19       event_buff != empty & step                             : 0;
20       1                                                      : step;
21     esac;
22   init(substate1) := nil;
23   next(substate1) :=
24     case
25       state = S_1 & event_buff = e_1 & gc & !step            : V_1;
26       substate1 = V_1 & event_buff = e_2 & cont_substate1    : V_2;
27       substate1 = V_2 & event_buff = e_3 & end_substate1     : nil;
28       substate1 = V_2 & event_buff = e_3 & cont_substate1    : V_2;
29       1                                                      : substate1;
30     esac;
31   ...
32   next(ack_substate1) :=
33     case
34       substate1 = V_1 & event_buff = e_2 & !cont_substate1 &
35       !superstate_enable                                     : neg;
36       substate1 = V_2 & event_buff = e_3 & (!cont_substate1 |
37       !end_substate1) & !superstate_enable                   : pos;
38       substate1 = next(substate1) & !step &
39       !superstate_enable & !substate1 = nil                  : neg;
40       1                                                      : undefine;
41     esac;
42   ...
```

Figure 6.14: NuSMV code for the concurrent composite state (lines 1–42)

```
43   init(cont_substate1) := 0;
44   next(cont_substate1) :=
45     case
46       state = S_2 & ack_substate1 = neg & ack_substate2 = neg
47       & !step & superstate_enable                              : 1;
48       state = S_2 & ack_substate1 = pos & ack_substate2 = neg
49       & !step & superstate_enable                              : 1;
50       state = S_2 & ack_substate1 = neg & ack_substate2 = pos
51       & !step & superstate_enable                              : 1;
52       state = S_2 & cont_substate1 & cont_substate2 & !step    : 0;
53       1                                                        : cont_substate1;
54     esac;
55   ...
56   init(end_substate1) := 0;
57   next(end_substate1) :=
58     case
59       state = S_2 & ack_substate1 = pos & ack_substate2 = pos
60       & !step & superstate_enable                              : 1;
61       state = S_2 & end_substate1 & end_substate2 & !step      : 0;
62       1                                                        : end_substate1;
63     esac;
64   ...
65   init(superstate_enable) := 0;
66   next(superstate_enable) :=
67     case
68       (next(ack_substate1) = pos |
69        next(ack_substate1) = neg) &
70       (next(ack_substate2) = pos |
71        next(ack_substate2) = neg) & !step                      : 1;
72       1                                                        : 0;
73     esac;
74   next(event_buff) :=
75     case
76       event_buff !=empty & next(step)                          : empty;
77       1                                                        : event_buff;
78     esac;
79 FAIRNESS running
```

Figure 6.15: NuSMV code for the concurrent composite state (lines 43–79)

chart diagrams. The temporal operator **X** is not preserved by the translation, whereas the operators **G**, **F** and **A** are preserved by the translation as illustrated by the CTL formula. Like the path quantifier **A**, the translation preserves the path quantifier **E** since the addition of intermediate states between two states of a statechart diagram only increases the number of states on a path and does not create any new (branching)

paths.

Theorem 15 *Given $F \in \mathcal{SC}, \beta(F)$ is its equivalent NuSMV representation, $CTL_{\{\mathbf{G},\mathbf{F},\mathbf{A},\mathbf{E}\}}$ denotes a subset of CTL with operators $\mathbf{G}, \mathbf{F}, \mathbf{A}$ and $\mathbf{E}$ and ψ is a formula of $CTL_{\{\mathbf{G},\mathbf{F},\mathbf{A},\mathbf{E}\}}$. If $F \models \psi$ then $\beta(F) \models \psi$.*

Proof sketch. Let $S_1, S_2 \in States(F)$, a transition connects S_1 and S_2, AP be a set of atomic propositions, p ranges over AP, S_{NuSMV} be a set of NuSMV states and $L : S_{NuSMV} \rightarrow 2^{\mathsf{AP}}$ be a labelling function that returns a set of atomic propositions which are true at a particular NuSMV state. Consider the following cases.

Case 1. *Operator $\mathbf{G}$. Suppose $F \models \mathbf{G}p$. Then there exists a sequence of transitions $\beta(S_1) \longrightarrow istate_1 \longrightarrow \ldots \longrightarrow istate_n \longrightarrow \beta(S_2)$ in the corresponding NuSMV representation such that $istate_i$ for $i = 1, \ldots, n$ model the run-to-completion step semantics, $p \in L(\beta(S_1))$ and $p \in L(\beta(S_2))$. Since p does not depend on $istate_i$ for $i = 1, \ldots, n$ which are unobservable in F, $p \in L(istate_i)$. Thus, if $F \models \mathbf{G}p$ then $\beta(F) \models \mathbf{G}p$.*

Case 2. *Operator $\mathbf{F}$. Suppose $F \models \mathbf{F}p$. Similarly, there exists a sequence of transitions $\beta(S_1) \longrightarrow istate_1 \longrightarrow \ldots \longrightarrow istate_n \longrightarrow \beta(S_2)$ in the corresponding NuSMV representation such that $istate_i$ for $i = 1, \ldots, n$ model the run-to-completion step semantics, $p \in L(\beta(S_1))$ and $p \in L(\beta(S_2))$. Since $p \in L(\beta(S_2))$, $\beta(F) \models \mathbf{F}p$. Thus, if $F \models \mathbf{F}p$ then $\beta(F) \models \mathbf{F}p$.*

Case 3. *Operator $\mathbf{A}$. Let $S_3 \in States(F)$ and a transition connects S_1 and S_3. Suppose $F \models \mathbf{AG}p$. Then there exists two sequences of transitions $\beta(S_1) \longrightarrow \ldots \longrightarrow \beta(S_2)$ and $\beta(S_1) \longrightarrow \ldots \longrightarrow \beta(S_3)$. Since the number of branching paths for both F and $\beta(F)$ is equal to 2 and $\mathbf{G}p$ holds for each of the sequence of transitions according to Case 1, $\beta(F) \models \mathbf{AG}p$. Thus, if $F \models \mathbf{AG}p$ then $\beta(F) \models \mathbf{AG}p$. The same argument can be used to prove that if $F \models \mathbf{AF}p$ then $\beta(F) \models \mathbf{AF}p$.*

Case 4. *Operator $\mathbf{E}$. Analogous to Case 3.*

Suppose $F \models \psi$. As a formula ψ is constructed from one or more atomic propositions and p is an arbitrary atomic proposition, it follows that $\beta(F) \models \psi$ holds. Thus, if $F \models \psi$ then $\beta(F) \models \psi$.

The desired property that an invalid state configuration, in which V_1 and W_2 are active, does not occur is specified as:

$$\mathbf{AG}!(concurr.substate1 = V\_1 \ \&\ concurr.substate2 = W\_2)$$

It took 1 second to verify these two formulas on a Pentium 4 2.4GHz PC with 512MB of memory using NuSMV 2.1.2 (with -dynamic and -f options) running under the Windows XP Professional operating system. This example illustrates the applicability of our approach for the analysis of concurrent composite states.

6.9 Summary

[65] has verified a variant of the SET protocol using the FDR model checker. We develop their idea further by studying the agent-based version of the SET protocol.

This chapter is a first step towards using an integrated approach for analyzing an agent-based protocol. We have tackled the problems which were raised at the beginning of this chapter. First, we have illustrated how an integrated approach was used for analyzing the non-security aspects of the SET/A protocol. Second, we have presented an extended SET/A protocol in which the cardholder and payment gateway can reach a consensus when an error occurs during the agent transmission process. Third, the extended SET/A protocol ensures that the non-faulty parties cardholder and payment gateway can agree on a decision even if an agent failure occurs after the arrival of the agent at the merchant's server. In addition, we have demonstrated the applicability of the approach for analyzing concurrent composite states.

Chapter 7

An Integrated Environment for Communicating UML Statechart Diagrams

As formal methods [77, 74, 30, 47, 73, 90, 106, 50] based on different underlying theories focus on different aspects of a system, providing an integrated environment for applying various well-established formal methods to analyze a specification expressed in a semi-formal graphical notation contributes to a more complete analysis.

An effort which integrates UML statechart diagrams and the π-calculus for checking equivalences of statechart diagrams is reported in Chapter 4. Similarly, an attempt which combines UML statechart diagrams, the π-calculus and the NuSMV model checker is described in Chapter 5. This chapter extends the work of previous chapters and presents an integrated environment (Figure 7.1) which unifies equivalence-checking and model-checking as a tool set.

The remainder of this chapter is organized as follows. Section 7.1 presents the design of the equivalence-checking environment and examines the implementation of the SC2PiCal translator. The use of the equivalence-checking environment is demonstrated by a number of examples and a case study. Section 7.2 gives the design of the model-checking environment and discusses the implementation of the PiCal2NuSMV translator. An evaluation of the model-checking environment using a case study approach is provided. The lessons learned from the applications of the integrated environment are described in Section 7.3. Previous work is reviewed in 7.4. Section 7.5 summarizes the

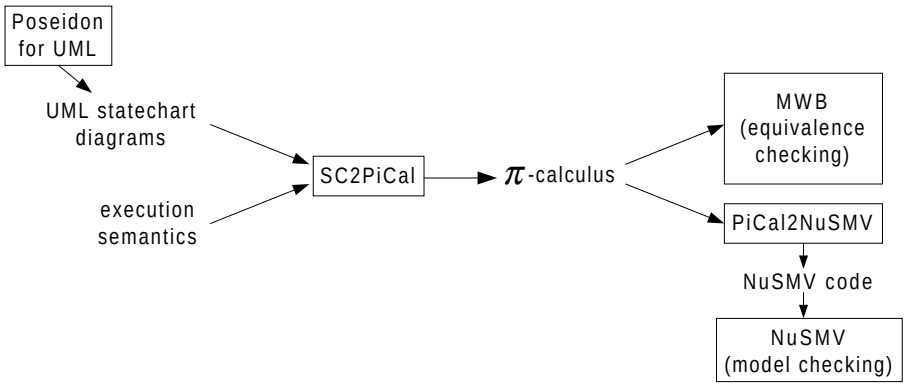

Figure 7.1: Overview of the integrated environment

chapter.

The material in this chapter is drawn from [52].

7.1 Automated Equivalence Checking

To provide an environment which supports the equivalence-checking of UML statechart diagrams, we have integrated two software tools: Poseidon for UML [13] and Mobility Workbench (MWB) [110, 111].

ArgoUML [94] is an extensible, open source UML modelling tool. It is implemented in Java [2] and is platform independent. Poseidon for UML, which is available in different editions, is a commercial variant of ArgoUML. The Community Edition of Poseidon for UML, like ArgoUML, is also a free UML modelling tool. It uses XML Metadata Interchange (XMI) 1.2 [41] as its storage format. XMI, which is an open standard adopted by Object Management Group (OMG), specifies how UML models are represented in Extensible Markup Language (XML) [18]. It permits the interoperability between UML modelling tools.

The MWB which is written in Standard ML (SML) [78] is a software tool for mobile processes. It runs under the Standard ML of New Jersey [91] and supports the analysis of concurrent systems which have dynamically changing interconnection structures. In particular, it allows the verification of strong and weak open bisimulations for processes specified in the π-calculus.

The architecture of our equivalence-checking environment is shown in Figure 7.2.

To integrate the two existing software tools, Poseidon for UML and MWB, we have developed a translator called SC2PiCal. The left arm, right arm and bottom leg of the T-diagram [7] specify the source language, target language and implementation language of the corresponding software tool with name shown below the T-diagram. The SC2PiCal, implemented in Java, translates an XMI document into an equivalent representation in the π-calculus.

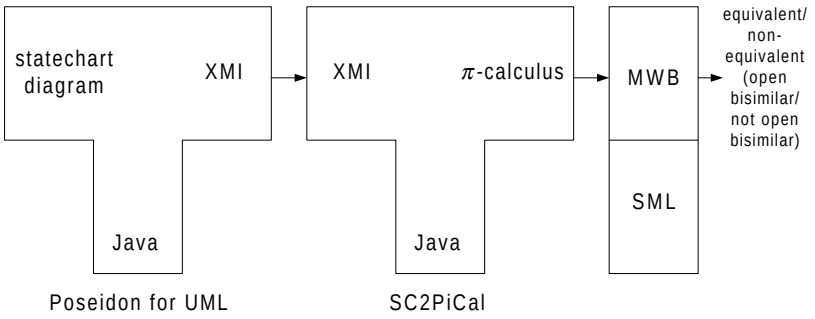

Figure 7.2: The equivalence-checking environment

To prove the equivalence of two UML statechart diagrams, the two diagrams are drawn and their corresponding XMI documents are generated using Poseidon. The SC2PiCal translator then transforms the XMI documents into π-calculus expressions (i.e. MWB code). Finally, the MWB checks whether the π-calculus expressions of the two statechart diagrams are open bisimilar or not.

7.1.1 The Implementation of the SC2PiCal

Since XMI is based on XML, an XMI document is just an XML document in which the XML elements conform to the XMI specification. Hence, XML Application Programming Interfaces (APIs) [1] such as Document Object Model (DOM) and Simple API for XML Processing (SAX) can both be used for creating, reading and processing an XMI document.

As the DOM and SAX APIs are contained in the Java API for XML Processing (JAXP) [1] which has incorporated into Java 2 Platform Standard Edition (J2SE) version 1.4.1 [108], we have adopted J2SE version 1.4.1 as our development tool for the SC2PiCal. In the implementation of the SC2PiCal, we use DOM (Level 2 API) rather than SAX as it provides a more flexible way for accessing the contents of an

XMI document.

The SC2PiCal (Figure 7.3) translates a statechart diagram expressed in XMI into a π-calculus representation specified in the input format of the MWB as three steps. First, the SC2PiCal reads an XMI document into a memory-resident DOM tree in which it represents the input XMI document. Second, the SC2PiCal traverses the DOM tree, extracts information related to states, events, guard-conditions, actions, etc. and stores them into a number of arrays, lists, maps and sets. Finally, the SC2PiCal generates the equivalent π-calculus representation based on the extracted information and the set of translation rules defined in Chapter 3.

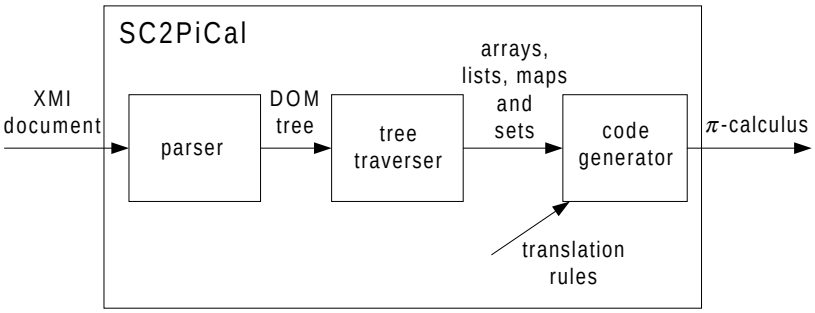

Figure 7.3: The architecture of the SC2PiCal translator

7.1.2 Using the Equivalence-checking Environment

In this subsection, we illustrate how the statechart diagrams which are classified into strong behaviour equivalence and weak behaviour equivalence in Chapter 4 are automatically proved using the equivalence-checking environment.

We first draw the statechart diagrams F_2 and G_2 in Figure 4.2 using Poseidon for UML Community Edition 1.6 and save them as two XMI documents. Then we translate the two XMI documents into their corresponding π-calculus representations as shown in Figures 7.4 and 7.5 using the SC2PiCal.

Note that the textual representations of $\boldsymbol{\nu}$ and $\overline{x}$ in the MWB are ^ and 'x, respectively. In addition, each process identifier definition in the MWB starts with the keyword *agent*. The MWB commands for importing the π-calculus representations of F_2 and G_2 (i.e. files $f_2.pi$ and $g_2.pi$) and checking the equivalence (congruence) of the statechart diagrams are shown in Figure 7.6. The MWB command *eq* checks whether

```
agent S1(step,event_ch0,e2,e1)= \
event_ch0(x1). \
([x1=e1] \
'step.S2(step,event_ch0,e2,e1)+ \
[x1=e2]'step.S1(step,event_ch0,e2,e1))

agent S2(step,event_ch0,e2,e1)= \
event_ch0(x1). \
([x1=e2] \
'step.S1(step,event_ch0,e2,e1)+ \
[x1=e1]'step.S2(step,event_ch0,e2,e1))
```

Figure 7.4: The π-calculus representation of F_2 (file $f_2.pi$)

```
agent T1(step,event_ch0,e2,e1)= \
event_ch0(x1). \
([x1=e1] \
'step.T2(step,event_ch0,e2,e1)+ \
[x1=e1]'step.T2(step,event_ch0,e2,e1)+ \
[x1=e2]'step.T1(step,event_ch0,e2,e1))

agent T2(step,event_ch0,e2,e1)= \
event_ch0(x1). \
([x1=e2] \
'step.T1(step,event_ch0,e2,e1)+ \
[x1=e1]'step.T2(step,event_ch0,e2,e1))
```

Figure 7.5: The π-calculus representation of G_2 (file $g_2.pi$)

two processes $P, Q \in \mathcal{A}$ are related by a strong open bisimulation.

Similarly, we prove that the statechart diagrams in Figure 4.4 (i.e. F_3 and G_3) and the statechart diagrams in Figure 4.7 (i.e. F_4 and G_4) are both equivalent. The following MWB command is used for analyzing F_3 and G_3:

$$weqd\ (step, event, e_1, e_2, e_3, pos, neg)\ S_3(step, event, e_2, e_1, e_3)$$

```
 The Mobility Workbench
 (MWB'97, polyadic version 3.122, built Mon Apr 21
  23:02:07 2003)

MWB>input "f2.pi"
MWB>input "g2.pi"
MWB>eq S1(step, event, e2, e1) T1(step, event, e2, e1)
The two agents are related.
Relation size = 4.
MWB>
```

Figure 7.6: Equivalence-checking of F_2 and G_2 with the MWB

Statechart Diagrams	Size of the Open Bisimulation Relation	Real Time Elapsed (secs)
F_2, G_2	4	0.000
F_3, G_3	14	0.016
F_4, G_4	36	0.032

Table 7.1: Performance analysis of equivalence checking

$T_2(step, event, e_2, e_1, e_3, pos, neg)$

The MWB command *weqd* checks whether the two processes with process identifiers $S_3(step, event, e_2, e_1, e_3)$,$T_2(step,event,e_2, e_1,e_3,pos,neg) \in \Im$ are related by a weak open bisimulation given that channels $step$,$event$,e_1,e_2,e_3,pos and neg are distinct. Table 7.1 shows the size of the open bisimulation relation $\mathcal{R}$ [110, 111] and the real time elapsed for equivalence checking in seconds for the three pairs of statechart diagrams. The real time elapsed for equivalence checking is obtained by using the MWB *time* command. The tests run under Windows XP on a 2.4GHz Pentium 4 PC with 512 MB of memory using MWB version 3.122.

7.1.3 Evaluation of the Equivalence-checking Environment

This subsection evaluates the equivalence-checking environment using the SET/A case study introduced in previous chapter. We first model the agent of the extended SET/A

protocol proposed in previous chapter as two different versions of statechart diagrams. They are then shown to be equivalent using the proposed equivalence-checking environment. In addition, the whole SET/A protocol which includes the cardholder (customer), the merchant, the agent and the payment gateway is also translated into the π-calculus using the SC2PiCal for verifying the generated π-calculus expressions based on the proposed translation rules discussed in Chapter 3.

To recapitulate, SET/A is an agent-based payment protocol which is designed for secure credit card payment in mobile computing environment. The agent used in the protocol is a mobile agent which travels from the cardholder's computer to the merchant's server. Upon arrival at the merchant's server, it first sends a purchase request to the merchant. Then it waits for a response from the merchant and travels back to the cardholder's computer. For a more detailed description of the protocol, the reader is referred to the previous chapter.

Figure 7.7 shows the agent as a flat statechart diagram with 9 non-composite states and 13 transitions. Through clustering, we represent the agent as a hierarchical statechart diagram (Figure 7.8) with 9 non-composite states, 2 non-concurrent composite states and 10 transitions as discussed in previous chapter. Out of the 10 transitions, 6 of them are interlevel transitions. To distinguish between the states of the flat and hierarchical statechart diagrams, suffixes 1 and 2 are added to the state names of the flat statechart diagram and hierarchical statechart diagram, respectively.

The model of the agent is a good candidate for testing the equivalence-checking environment. First, it contains the three basic interlevel transition patterns as shown in Figure 7.9. Second, in Figures 7.7 and 7.8 the action parts of the transitions are *send* actions in which the *agent* object sends events to target objects *cardholder* and *merchant*. This illustrates that the equivalence-checking environment supports communication between multiple interacting statechart diagrams in which they are associated with multiple objects.

We translate the two versions of the statechart diagrams into the π-calculus using the SC2PiCal. They are shown to be equivalent using the MWB. The size of the open bisimulation relation is 67. It took 1.281 seconds of real time to perform the equivalence-checking process.

To provide a more detailed analysis of the equivalence-checking environment, we further investigate the relationships between:

(i) the complexity of the pair of statechart diagrams and the time taken for the

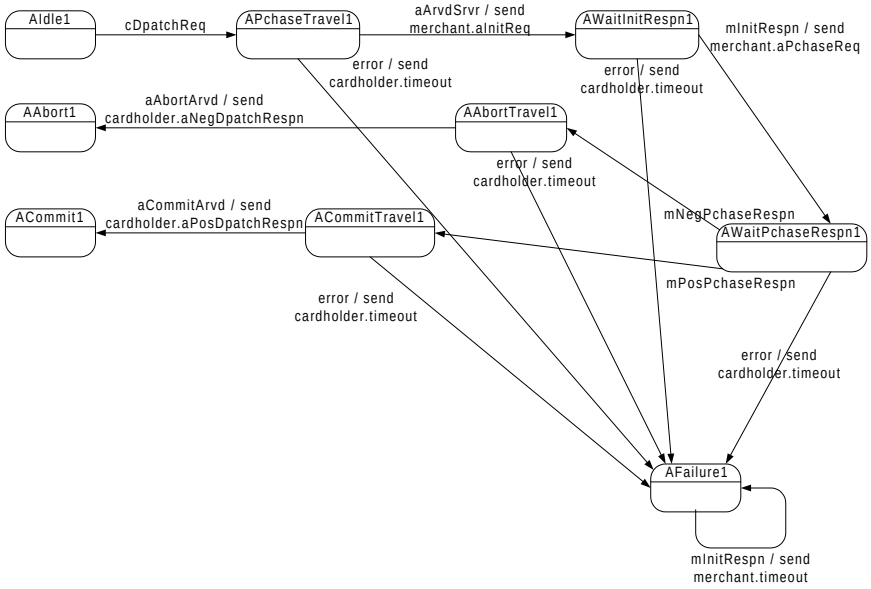

Figure 7.7: Flat statechart diagram for the agent

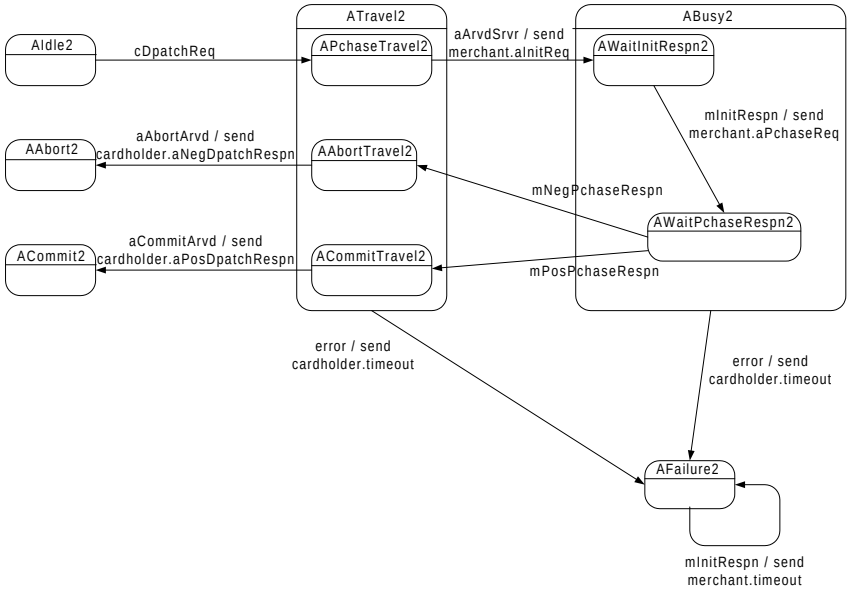

Figure 7.8: Hierarchical statechart diagram for the agent

translations by the SC2PiCal translator;

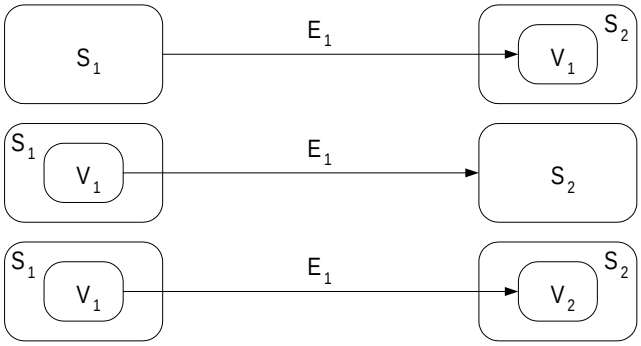

Figure 7.9: Interlevel transition patterns

(ii) the complexity of the pair of statechart diagrams and the size of open bisimulation relation; and

(iii) the complexity of the pair of statechart diagrams and the real time elapsed for equivalence checking by the MWB.

A number of factors which contribute to the complexity of the statechart diagrams are identified and measured as shown in Table 7.2. The number of send actions is a measure of the degree of interaction between communicating statechart diagrams.

As illustrated in the table, the increase in the complexity of the pair of statechart diagrams does not have much effect on the time taken for the translations by the SC2PiCal translator. In contrast, the increase in the complexity of the pair of statechart diagrams leads to an increase in the size of open bisimulation relation and real time elapsed for equivalence checking by the MWB. To take a closer look at the relationship between (i) the complexity of the pair of statechart diagrams and the size of open bisimulation relation; and (ii) the complexity of the pair of statechart diagrams and the real time elapsed for equivalence checking, we increase the size of the number of substates in the hierarchical statechart diagram and number of states in the unfolded version of the statechart diagram in Figure 4.6. Table 7.3 presents the results of performance tests. A plot of (i) size of open bisimulation relation against number of substates/states; and (ii) real time elapsed for equivalence checking against number of substates/states are given in Figures 7.10 and 7.11. The size of open bisimulation relation is a linear function of the number of substates in the hierarchical statechart diagram/number of states in the unfolded version of the statechart diagram. In contrast,

	Statechart Diagrams							
	Pair 1		Pair 2		Pair 3		Case Study	
	F_2	G_2	F_3	G_3	F_4	G_4	Flat agent	Hierarchical agent
Complexity								
no. of basic states	2	3	3	3	5	5	9	9
no. of non-concurrent composite states	0	0	0	1	0	0	0	2
no. of concurrent composite states	0	0	0	0	1	0	0	0
total no. of states	2	3	3	4	6	5	9	11
no. of non-interlevel transitions	2	4	4	2	2	5	13	4
no. of interlevel transitions	0	0	0	1	0	0	0	6
no. of incoming transitions to fork pseudostates	0	0	0	0	1	0	0	0
no. of outgoing transitions from fork pseudostates	0	0	0	0	2	0	0	0
total no. of transitions	2	4	4	3	5	5	13	10
no. of send actions	0	0	0	0	0	0	10	7
size of open bisimulation relation	4		14		36		67	
Performance								
time taken for the translations by the SC2PiCal (secs)	< 1		< 1		< 1		< 1	
real time elapsed for equivalence checking by the MWB (secs)	0.000		0.016		0.032		1.281	
total time taken (secs)	< 1.000		< 1.016		< 1.032		< 2.281	

Table 7.2: Performance analysis of the equivalence-checking environment

no. of substates in the hierarchical statechart diagram / no. of states in the unfolded version of the statechart diagram	size of open bisimulation relation	real time elapsed for equivalence checking by the MWB
8	73	0.641
16	113	2.328
32	193	17.844
48	273	76.296
64	348	235.796

Table 7.3: Variation of size of open bisimulation relation and real time elapsed for equivalence checking against no. of substates/states

the real time elapsed for equivalence checking is an exponential function of the number of substates in the hierarchical statechart diagram/number of states in the unfolded version of the statechart diagram.

7.2 Automated Model Checking

To determine whether a system represented in the π-calculus satisfies its specifications, a model-checking environment (Figure 7.12) is provided. The model-checking environment consists of two components: PiCal2NuSMV and NuSMV.

The PiCal2NuSMV, which is written in Java, translates π-calculus expressions generated by the SC2PiCal into the NuSMV input language in a single pass.

The NuSMV model checker is implemented in ANSI C. It verifies the model specified by the generated NuSMV code against the Computation Tree Logic (CTL) specifications. If the model is invalid, a counter-example is generated for each violated CTL specification.

7.2.1 The Implementation of the PiCal2NuSMV

The transformation from the π-calculus into the NuSMV input language (Figure 7.13) is divided into three stages: lexical analysis, syntax analysis and code generation.

The lexer inputs a stream of characters from a file and groups them into π-calculus tokens. The parser determines whether a stream of π-calculus tokens conform to the

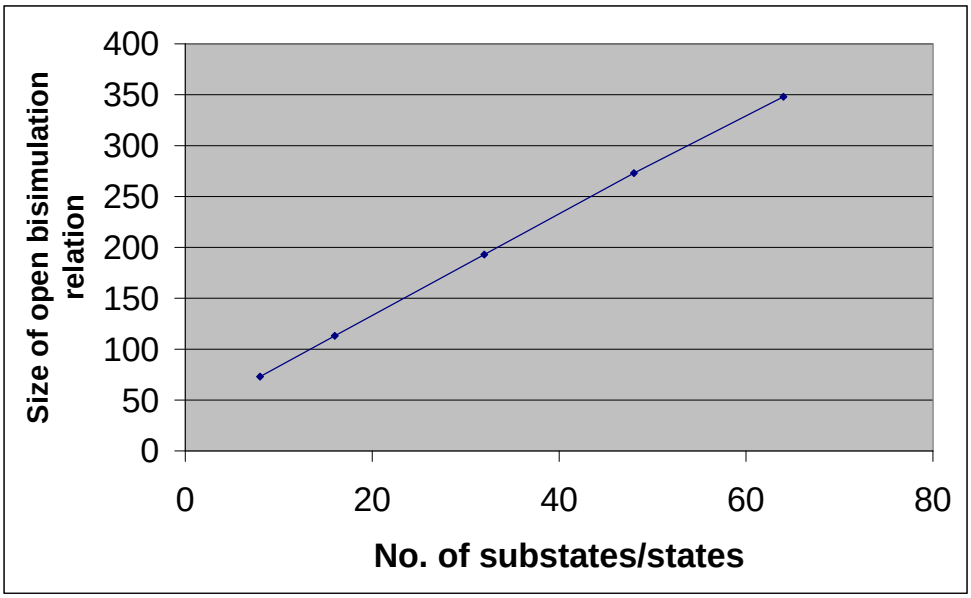

Figure 7.10: Size of open bisimulation relation vs. no. of substates/states

π-calculus grammar. It collects and stores information related to module declarations, variable declarations, case expressions, etc. into a number of lists and sets. The code generator then produces NuSMV code using the translation rules defined in Chapter 5.

The lexer and parser are constructed using a compiler tool ANTLR (ANother Tool for Language Recognition) 2.7.2 [86]. We define π-calculus lexer rules in ANTLR syntax which is based on regular expressions. Similarly, we specify π-calculus parser rules in ANTLR syntax which is based on Extended Backus-Naur Form (EBNF). As the lexer and parser rules are stored in a single file, we execute the Java class *antlr.Tool* which inputs the grammar file and generates the lexer and parser in Java. The generated Java source code is then compiled into Java bytecode as usual.

The code generator, which is written in Java, is built by hand. It implements the translation rules and outputs NuSMV source code according to the collected compilation information in the syntax analysis stage.

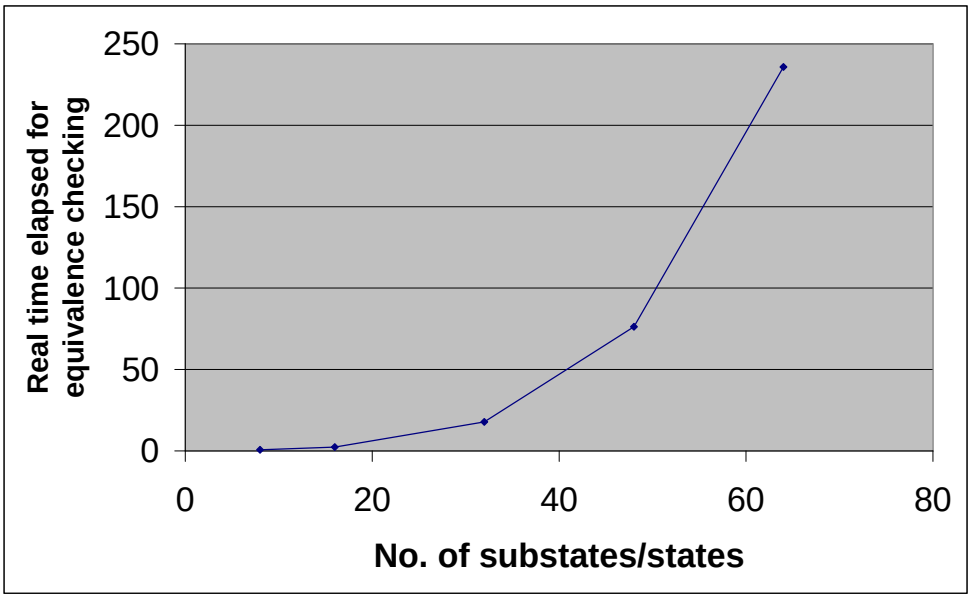

Figure 7.11: Real time elapsed for equivalence checking vs. no. of substates/states

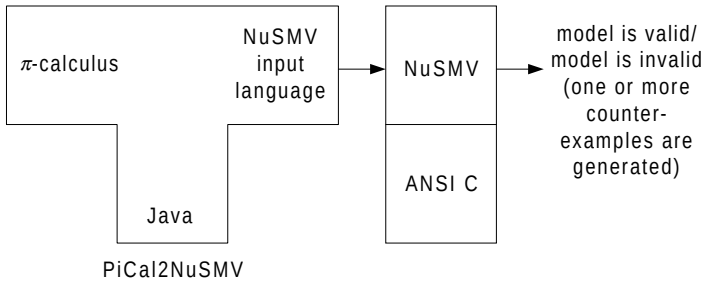

Figure 7.12: The model-checking environment

7.2.2 Using the Model-checking Environment

We first use Poseidon for UML to document the behaviour of a system represented using several (linked) statechart diagrams. Then we save the statechart diagrams into XMI documents and translate the XMI documents into equivalent π-calculus representations

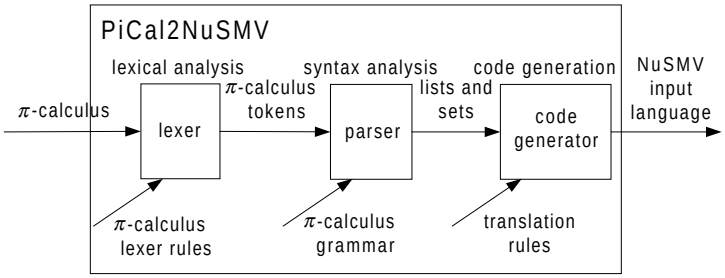

Figure 7.13: The architecture of the PiCal2NuSMV translator

which are stored in separate files using the SC2PiCal described in previous section. The PiCal2NuSMV reads a text file which contains all these file names, translates each file into NuSMV code and generates the equivalent representation of the system in the NuSMV input language as a single file. NuSMV inputs the generated file, checks the model against the CTL formulas specified by the user and constructs a trace when an CTL formula is invalid.

7.2.3 Evaluation of the Model-checking Environment

Two case studies are used for evaluating the model-checking environment. The first case study is based on the original SET/A protocol, whereas the second case study is based on the extended version of the SET/A protocol proposed in Chapter 6. Unlike the original SET/A protocol, the extended SET/A protocol ensures that a transaction is resilient to the failure of the mobile shopping agent.

We model the cardholder, agent, merchant and payment gateway of the original SET/A protocol as 4 statechart diagrams with 23 states and 17 transitions. Out of the 23 states and 17 transitions, there are 2 non-concurrent composite states and 6 interlevel transitions (see Table 7.4). Then we translate the statechart diagrams into their π-calculus equivalent representations as 23 process definitions using the SC2PiCal. Finally, the 23 process definitions are transformed into the NuSMV input language as 4 modules using the PiCal2NuSMV.

The translations from statechart diagrams into π-calculus specifications using SC2PiCal and π-calculus specifications into NuSMV code using PiCal2NuSMV took 2 seconds and 1 second, respectively. The verification of the three CTL formulas discussed in Chap-

	Case Studies	
	SET/A Protocol	Extended SET/A Protocol
Complexity		
no. of basic states	21	24
no. of non-concurrent composite states	2	2
total no. of states	23	26
no. of non-interlevel transitions	11	25
no. of interlevel transitions	6	6
total no. of transitions	17	31
no. of send actions	11	21
Performance		
time taken for the translation by the SC2PiCal (secs)	2	2
time taken for the translation by the PiCal2NuSMV (secs)	1	1
time taken for the verification by the NuSMV (secs)	4	23
no. of BDD nodes allocated	87897	222855
total time taken (secs)	7	26

Table 7.4: Performance analysis of the model-checking environment

ter 6 took approximately 4 seconds on a Pentium 4 2.4GHz PC with 512MB of memory using NuSMV 2.1.2 (with -dynamic and -f options) running under the Windows XP Professional operating system.

A similar exercise is carried out with the extended SET/A protocol. In the statechart diagrams of the extended SET/A protocol, there are 26 states and 31 transitions. Out of the 26 states and 31 transitions, there are 2 non-concurrent composite states and 6 interlevel transitions. The translations took 2 seconds and 1 second, respectively. The desired properties are satisfied by the extended SET/A protocol and the verification took 23 seconds. The results are presented in Table 7.4.

As Table 7.4 shows, the increase in the numbers of basic states, non-interlevel transitions and send actions by factors of 1.14, 2.27 and 1.91, respectively, does not have much effect on the translation times of the SC2PiCal and PiCal2NuSMV translators. However, the number of BDD nodes allocated increases by a factor of 2.54. The verification time taken by the NuSMV model checker increases by a factor of 5.75 in which the order of magnitude is quite close to the theoretical value if the weightings of all complexity factors are assumed to be 1. A more detailed treatment of the performance of NuSMV in terms of BDDs by using a number of examples described in previous studies is provided in [27].

7.3 Lessons Learned

The lessons gained from the use and evaluation of the integrated environment are enumerated as follows:

1. We assume that the weightings of all complexity factors in Tables 7.2 and 7.4 are equal to the value 1. Consider the hierarchical agent and the SET/A protocol, the overall complexity of statechart diagrams increases by a factor of 10.06 when the numbers of basic states, non-interlevel transitions and send actions increase by factors of 2.33, 2.75 and 1.57, respectively. As the time taken for the translation by the SC2PiCal translator increases by only a factor of around 2, this reveals that the SC2PiCal translator has a relatively small effect on the performance of the integrated environment.
2. As discussed in the previous section, the increase in the complexity of statechart diagrams does not have much effect on the translation times of the PiCal2NuSMV translator.

3. The execution times for equivalence checking and model checking increase as the complexity of statechart diagrams increases. As illustrated in Figure 7.11, the growth rate of real time elapsed for equivalence checking is exponential. The performance of the equivalence-checking environment and model-checking environment depends mainly on the software tools MWB and NuSMV rather than on the translators SC2PiCal and PiCal2NuSMV.
4. To address the state explosion problem of the MWB and NuSMV, the essence is to reduce a problem to a manageable size through (i) adopting abstraction technique [30, 49] that abstracts away irrelevant details; and (ii) performing formal analysis on the critical part of a complex system instead of the complete system.

7.4 Related Work

In [105, 17] a formalism FOCUS, that supports different views for the specification of embedded systems, is integrated with various formal methods and tools for model checking, theorem proving and testing. In contrast, our integration of formal methods and tools combines UML statechart diagrams with the π-calculus, MWB and NuSMV for equivalence checking and model checking. When compared with FOCUS, our approach provides a wider coverage as it is not only limited to embedded systems.

7.5 Summary

The integrated environment comprises a collection of heterogeneous tools for the analysis of UML statechart diagrams. Two translators SC2PiCal and PiCal2NuSMV have been developed for combining Poseidon for UML, MWB and NuSMV as an integrated environment which supports equivalence checking and model checking. The SC2PiCal transforms statechart diagrams represented as XMI documents into π-calculus specifications, while the PiCal2NuSMV generates NuSMV code from the π-calculus specifications.

The correctness and performance of the integrated environment have been tested and evaluated using the original SET/A protocol and the extended SET/A protocol. As the SET/A protocol is based on the SET protocol which is a real-world application, this proves that the integrated environment works in practice.

The integrated environment is novel in the sense that it combines various well-

established formal methods and tools as a tool set for the analysis of UML statechart diagrams. The novel features of the integrated environment are summarized below:

1. The proposed integrated environment is a cross-disciplinary challenge. Methods and software tools originally used in process algebras and model checking are adapted to the software engineering field for analyzing and verifying the design of an agent-based e-commerce payment protocol.
2. The integrated environment not only allows two analysis tools to be executed from a single interface, but it provides new capabilities that facilitate equivalence-checking and model-checking of UML statechart diagrams.
3. The architecture of the integrated environment is based on XMI that permits the exchange of data between UML modelling tools.
4. The integrated environment adopts an open, extensible and modular architecture. A new tool can be easily incorporated into the existing integrated environment through the development of a mapping between the π-calculus and the new tool.

Chapter 8

Conclusions

The lack of a formal execution semantics and the need for a formal analysis of UML statechart diagrams motivate this work. A precise execution semantics is a prerequisite for carrying out any form of formal analysis on UML statechart diagrams. The formalization is a difficult task as (i) the UML documentation is incomplete; and (ii) the behaviour of an interlevel transition is hard to represent. When compared with traditional testing techniques, a formal analysis provides a thorough analysis of a model against its specifications. It allows the verification of the model in the design stage of software development instead of the implementation stage.

This thesis has presented improvements and extensions to the original execution semantics. An attempt to formalize the execution semantics in the π-calculus has been described. Examples illustrating the formalization of various graphical constructs in the π-calculus have been provided. An integrated approach for equivalence checking and model checking of statechart diagrams has been proposed. To support the integrated approach, an integrated environment which offers tools for automating the equivalence-checking and model-checking processes has been implemented. The application of the integrated approach and integrated environment has been demonstrated by a case study.

8.1 Summary of Contributions

Our work contributes to the software engineering field by providing a precise definition for the execution semantics of communicating UML statechart diagrams and developing a systematic approach which unifies statechart diagrams with different formal methods and software tools. The main contributions of this thesis are enumerated as

the following points:

1. We have proposed a compositional approach for modelling state hierarchy through the use of parallel composition and firing priority scheme via channel passing. The structure-preserving approach greatly simplifies the encoding of interlevel transitions.
2. The execution semantics of communicating UML statechart diagrams has been formalized in the π-calculus. The formalization not only clarifies and extends the original UML semantics, but it provides a basis for the formal analysis of the statechart diagrams.
3. A rigorous approach, which is based on the structural congruence and open bisimulations of the π-calculus, has been presented. It provides a systematic way for proving the equivalence of two statechart diagrams.
4. An integration of statechart diagrams, the π-calculus and NuSMV for verifying the design of a system against its specifications has been advocated.
5. We have illustrated the applicability of the integrated approach by using the SET/A protocol as a case study. An examination reveals that the SET/A protocol is not resilient to the failure of the mobile agent. To ensure that all non-faulty parties agree on a same decision, an extended SET/A protocol has been given.
6. As a proof of concepts, an integrated environment which supports equivalence checking and model checking has been implemented. The existing tools Poseidon for UML, MWB and NuSMV are linked up through the two translators SC2PiCal and PiCal2NuSMV which we have developed.
7. The integrated environment automates the equivalence checking and model checking of UML statechart diagrams. It enables a user to perform a formal analysis and reasoning on the design of a system.

Although there are a number of studies on the formalization, equivalence checking and model checking of UML statechart diagrams [61, 60, 38, 39, 62, 113] in the literature, our approach is better as (i) state hierarchy and interlevel transitions are encoded in a simple and direct way; (ii) the formalizations of notational elements and execution semantics are both represented by the same formalism; (iii) the formalized execution semantics covers a single statechart diagram as well as multiple interacting statechart diagrams; (iv) the π-calculus is an intermediate representation which facilitates the equivalence checking of statechart diagrams; (v) the equivalence checking of statechart diagrams which contain interlevel transitions is addressed; and (vi) NuSMV allows

specifications to be expressed in either linear or branching temporal logic.

The main drawback of our approach is it is only technically limited to a subset of UML statechart diagrams. However, the proposed formalization is flexible enough for including other notational elements such as history pseudostates, state references and deferred events as discussed in Section 3.6.

Formal arguments on the correctness of the formalization of UML statechart diagrams and implementation of UML statecharts based π-calculus expressions in NuSMV language have been given. We have also defined isomorphism, strong behavioural equivalence and weak behavioural equivalence in terms of UML statechart diagrams and shown that there are a correspondence between isomorphism and structural congruence, strong behavioural equivalence and strong open bisimulation as well as weak behavioural equivalence and weak open bisimulation.

8.2 Future Research

This work is a first step towards equivalence checking and model checking of UML statechart diagrams using an integrated approach. Based on our current work, there are several directions for future research.

Firstly, we consider only the core part of statechart diagrams in our formalization. A future research direction is to formalize other notational elements such as time events and deferred events. The support of time events is of particular interest as it facilitates the analysis of real-time systems. To model time events, an extension to the π-calculus for encoding a timer is required. A detailed discussion on how a timer can be incorporated into the π-calculus is given in [8]. Another possible direction is to study the formalization of sequence diagrams. In the UML, the dynamic behaviour of a system is represented using various diagrams including statechart diagrams and sequence diagrams. A statechart diagram specifies the complete lifecycle of an object, while a sequence diagram specifies the partial lifecycle of each object which takes part in an interaction. Due to the existence of two model views, inconsistency between statechart diagrams and sequence diagrams occurs inevitably during the software development and software evolution processes. To ensure the inter-model consistency is preserved, an approach which is similar to the equivalence checking of statechart diagrams can be developed for checking whether a set of sequence diagrams is consistent with a corresponding set of statechart diagrams.

Secondly, the integrated approach is limited to equivalence checking and model checking of statechart diagrams. One promising research area is to examine intra-model consistency checking for determining whether the statechart diagrams of classes linked with a generalization relationship are consistent. Another promising research area is to explore the use of theorem proving [40] for the analysis of statechart diagrams.

Thirdly, the incorporation of other formalisms and software tools into the existing integrated environment for providing a wider range of analysis is also an interest research direction. In future work, we plan to incorporate more tools such as LySa Extractor [21] into our approach so that it provides a way to analyze the security aspects of an agent-based payment protocol.

Finally, more experiments on the integrated approach and integrated environment with respect to practicality, performance and learning curve are required before a technology transfer to industry is possible.

Appendix A

Command Reference for the Integrated Environment

The integrated environment includes a collection of tools which aim to support the equivalence checking and model checking of UML statechart diagrams, through the implementation of two translators SC2PiCal and PiCal2NuSMV and the use of three pre-existing tools Poseidon for UML, MWB and NuSMV which are all integrated for the specification and analysis of statechart diagrams.

This appendix contains a list of commands used for the equivalence checking and model checking of statechart diagrams. A description of the command's function is followed by the command syntax and options (if any). The conventions adopted in this appendix are given below:

1. Commands and options that are in boldface should be typed verbatim.
2. Options enclosed in square brackets are optional. Do not include the square brackets in your command.
3. Parameters enclosed in angle brackets should be replaced with user-specified values. Do not include the angle brackets in your command.
4. Parameter that can be repeated in a command is indicated by an ellipsis. Do not include the ellipsis in your command.
5. The blank spaces, double quotes, parentheses and commas are part of the syntax and require to be entered verbatim.

A.1 SC2PiCal

SC2PiCal is an academic tool which transforms a UML statechart expressed in XMI format into an equivalent π-calculus representation. It uses DOM (Level 2 API) integrated in Java 2 Platform Standard Edition (J2SE) for processing the XMI representation of the UML statechart.

java SC2PiCal – Translates a UML statechart diagram in XMI format into a corresponding representation in the π-calculus.

java SC2PiCal `<input-file> <output-file>`

A.2 MWB

MWB is the main tool for analyzing π-calculus processes. One of its key features is it permits the verification of open bisimulations for π-calculus processes. The tool is freely available on the Internet at

http://www.it.uu.se/research/group/mobility/mwb

mwb – invoke the MWB

mwb

input – reads π-calculus specifications from the specified file into the MWB

input "`<file>`"

eq – checks whether two processes are related by a strong open bisimulation

eq `<process1> <process2>`

weq – checks whether two processes are related by a weak open bisimulation

weq `<process1> <process2>`

eqd – checks whether two processes are related by a strong open bisimulation given that a list of channels separated by commas are distinct

eqd `(<channel1>,<channel2>,...,<channeln>) <process1> <process2>`

weqd – checks whether two processes are related by a weak open bisimulation given that a list of channels separated by commas are distinct

weqd `(<channel1>,<channel2>,...,<channeln>) <process1> <process2>`

time – executes a command and shows its execution statistics

time `<command>`

quit – exits the MWB

quit

Figure A.1 shows a sample session illustrating the verification of two UML statecharts based π-calculus processes (Figures A.2, A.3 and A.4) which are weakly open bisimilar.

```
 The Mobility Workbench
 (MWB'97, polyadic version 3.122, built Mon Apr 21 23:02:07 2003)

MWB>input "f3.pi"
MWB>input "g3.pi"
MWB>weqd (step,event,e1,e2,e3,pos,neg) S3(step,event,e2,e1,e3) \
T2(step,event,e2,e1,e3,pos,neg)
The two agents are related.
Relation size = 14.
MWB>
```

Figure A.1: A sample session of the MWB

```
agent S1(step,event_ch0,e2,e1,e3)= \
event_ch0(x1). \
([x1=e1] \
'step.S2(step,event_ch0,e2,e1,e3)+ \
[x1=e2]'step.S3(step,event_ch0,e2,e1,e3)+ \
[x1=e3]'step.S1(step,event_ch0,e2,e1,e3))

agent S2(step,event_ch0,e2,e1,e3)= \
event_ch0(x1). \
([x1=e2] \
'step.S3(step,event_ch0,e2,e1,e3)+ \
[x1=e1]'step.S2(step,event_ch0,e2,e1,e3)+ \
[x1=e3]'step.S2(step,event_ch0,e2,e1,e3))
```

Figure A.2: File content of *f3.pi*

```
agent S3(step,event_ch0,e2,e1,e3)= \
event_ch0(x1). \
([x1=e3] \
'step.S1(step,event_ch0,e2,e1,e3)+ \
[x1=e2]'step.S3(step,event_ch0,e2,e1,e3)+ \
[x1=e1]'step.S3(step,event_ch0,e2,e1,e3))
```

Figure A.3: File content of *f3.pi* (continued)

```
agent V1(event_ch1,e2,e1,e3,pos,neg)= \
event_ch1(x1,ack). \
([x1=e1] \
'ack<neg>.V2(event_ch1,e2,e1,e3,pos,neg)+ \
[x1=e2]'ack<pos>.0+ \
[x1=e3]'ack<neg>.V1(event_ch1,e2,e1,e3,pos,neg))

agent V2(event_ch1,e2,e1,e3,pos,neg)= \
event_ch1(x1,ack). \
([x1=e2] \
'ack<pos>.0+ \
[x1=e1]'ack<neg>.V2(event_ch1,e2,e1,e3,pos,neg)+ \
[x1=e3]'ack<neg>.V2(event_ch1,e2,e1,e3,pos,neg))

agent T1(step,event_ch0,event_ch1,e2,e1,e3,pos,neg)= \
event_ch0(x1). \
(^ack)'event_ch1<x1,ack>.ack(x2). \
([x2=pos]'step.T2(step,event_ch0,e2,e1,e3,pos,neg)+ \
 [x2=neg]'step.T1(step,event_ch0,event_ch1,e2,e1,e3,pos,neg))

agent T2(step,event_ch0,e2,e1,e3,pos,neg)= \
event_ch0(x1). \
([x1=e3] \
'step.(^event_ch1)(T1(step,event_ch0,event_ch1,e2,e1,e3,pos,neg) | V1(event_ch1,e2,e1,e3,pos,neg))+ \
[x1=e2]'step.T2(step,event_ch0,e2,e1,e3,pos,neg)+ \
[x1=e1]'step.T2(step,event_ch0,e2,e1,e3,pos,neg))
```

Figure A.4: File content of *g3.pi*

A.3 PiCal2NuSMV

PiCal2NuSMV is an academic tool which transforms UML statecharts based π-calculus representations into equivalent NuSMV code. The lexer and parser generated by ANTLR are used for parsing the UML statecharts based π-calculus expressions.

java PiCal2NuSMV – Transforms separate π-calculus representations of UML stat-

echart diagrams into corresponding NuSMV representations and combines them as a single file for model checking

java PiCal2NuSMV `<input-file> <output-file>`

where `<input-file>` is a plain text file which contains the filenames of separate π-calculus representations.

A.4 NuSMV

NuSMV is a model checker which checks whether a system fulfills its specifications. The tool is available free of charge on the Internet at

http://nusmv.irst.itc.it

nusmv – invokes NuSMV in batch mode

nusmv `[options] <input-file>`

useful options:

-dynamic	enables the use of dynamic variable ordering
-f	enables the computation of reachable states for improving the performance of CTL model checking

A sample session, which illustrates the verification of the two CTL formulas discussed in Section 6.8, is shown in Figure A.5.

```
C:\research\nusmv-2.1.2\bin>nusmv -dynamic -f concurr.smv
*** This is NuSMV 2.1.2 (compiled 2002-11-22 12:00:00)
*** For more information of NuSMV see <http://nusmv.irst.itc.it>
*** or email to <nusmv-users@irst.itc.it>.
*** Please report bugs to <nusmv-users@irst.itc.it>.
-- specification AG ((concurr.substate1 = V_1 & event_buff = e_2) -> AF concurr.
substate1 = V_2) is true
-- specification AG (!(concurr.substate1 = V_1 & concurr.substate2 = W_2)) is tr
ue
```

Figure A.5: A sample session of NuSMV

Appendix B

The Implementation of the Integrated Environment

Sections 7.1.1 and 7.2.1 give an overview of the implementation and architecture of the SC2PiCal and PiCal2NuSMV translators. The usage and command reference of the integrated environment are provided in Sections 7.1.2 and 7.2.2 as well as Appendix A. The practicality and performance of the integrated environment are evaluated using a case study approach in Sections 7.1.3 and 7.2.3. In addition, the applications of the integrated environment for the equivalence checking and model checking of UML statechart diagrams are demonstrated. This appendix expands on these previous sections by providing a more detailed discussion about the principles, concepts and techniques for the implementation of the SC2PiCal and PiCal2NuSMV using the Java programming language. The complete toolchain is illustrated through a number of examples.

B.1 The Architecture of the Integrated Environment

The integrated environment automates the equivalence checking and model checking of UML statechart diagrams. As shown in Figure B.1, the integrated environment consists of five components:

1. Poseidon for UML,
2. MWB (Mobility Workbench),
3. NuSMV,
4. SC2PiCal and

5. PiCal2NuSMV.

Poseidon for UML is a modelling tool for drawing UML statechart diagrams and generating the corresponding UML statechart diagrams in XMI format. MWB is a π-calculus tool which checks whether two π-calculus processes are strongly and weakly open bisimilar. NuSMV is a model checker which verifies whether a property is maintained or not in the specification. SC2PiCal is a translator which transforms UML statechart diagrams in XMI format into the π-calculus, whereas PiCal2NuSMV is a translator which transforms the UML statecharts based π-calculus into NuSMV code. Both translators are implemented in Java as (i) it is portable and (ii) it encompasses classes for manipulating an XMI (XML) document.

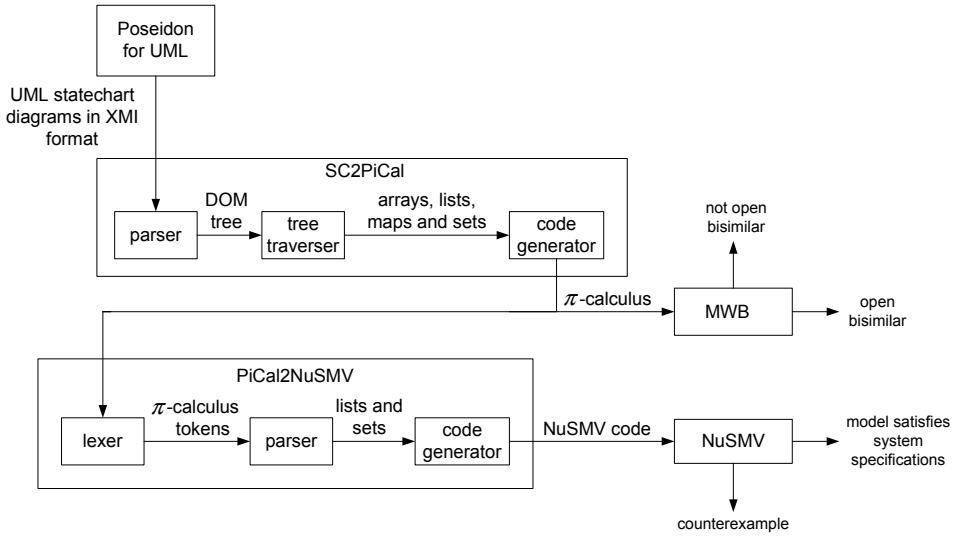

Figure B.1: Toolset architecture

B.2 The Detailed Implementation of SC2PiCal

The JAXP (Java API for XML Processing), which is a component of J2SE (Java 2 Platform Standard Edition), contains four APIs: DOM Level 1 and Level 2 and SAX1 and SAX2, but each of the first versions is now deprecated.

As pointed out in Section 7.1.1, we adopt DOM in preference to SAX2 because of the greater flexibility in manipulating an XML document as a result of the DOM parser constructing a memory-resident tree.

The three main phases of translating UML statechart diagrams in XMI format

into the π-calculus are delineated in Figure B.1. The first phase accepts an XMI document as input and generates a DOM tree using the APIs of DOM Level 2. The second phase traverses the DOM tree and stores each transition of a statechart diagram which includes source state, event, guard-condition, action and target state as elements of a 2-dimensional array. The final phase then produces the equivalent π-calculus specification (MWB code). The implementation starts with the use of standard patterns for constructing a DOM parser and acquiring the root of the DOM tree as follows:

```
1 DocumentBuilderFactory factory =
2    DocumentBuilderFactory.newInstance();
3 factory.setValidating( false );
4 factory.setNamespaceAware( true );
5 DocumentBuilder builder = factory.newDocumentBuilder();
6 builder.setErrorHandler( new MyErrorHandler() );
7 document = builder.parse( new File( file ) );
8 processNode( document );
```

The purpose of lines 1–5 is to produce a non-validating, namespace-aware DOM parser from an instance of the factory class *DocumentBuilderFactory*. A non-validating parser is fabricated instead of a validating one as the XMI document generated by Poseidon for UML is not associated with an XML schema or a Document Type Definition (DTD). The static method *newInstance* creates a new *DocumentBuilderFactory* object. The methods *setValidating* and *setNamespaceAware* specify that the *DocumentBuilder* object (parser) to be created performs no validation on the XMI (XML) document and treats a colon character as a delimiter between a namespace name and a tag name, respectively. The method *newDocumentBuilder* creates a *DocumentBuilder* object. The parameter of the method *setErrorHandler* defines that a *MyErrorHandler* object is used as the error handler. The method *parse* parses an XMI (XML) document and returns a *Document* object which is the root of the DOM tree. The returned *Document* object is then passed as a parameter to the method *processNode* (a tree traverser).

The tree traverser is implemented as two methods *processNode* and *processChildNodes* as shown in Figures B.2, B.3, B.4, B.5, B.6 and B.7. The major considerations and difficulties for the implementation of the tree traverser are:

1. The identifier (*id* attribute) is an unique identity for an XML element within an XMI document. As the XMI representation of a transition is based on identifiers rather than the names of source state, event, guard-condition, action and target state, there is a need to maintain a mapping between the identifiers and the names

```
1  public void processNode( Node currentNode )
2  {
3    String state_name="";
4    switch ( currentNode.getNodeType() ) {
5      case Node.DOCUMENT_NODE:
6        Document doc = ( Document ) currentNode;
7        processChildNodes( doc.getChildNodes() );
8        break;
9      case Node.ELEMENT_NODE:
10       if (currentNode.getNodeName().equals("UML:SimpleState") |
11         (currentNode.getNodeName().equals("UML:Transition") &
12          currentNode.getParentNode().getNodeName().equals("UML:StateMachine.transitions")) |
13         currentNode.getNodeName().equals("UML:Guard") |
14         currentNode.getNodeName().equals("UML:BooleanExpression") |
15         currentNode.getNodeName().equals("UML:CallAction") |
16         currentNode.getNodeName().equals("UML:ActionExpression") |
17         currentNode.getNodeName().equals("UML:CallEvent") |
18         (currentNode.getNodeName().equals("UML:CompositeState") &
19          currentNode.getParentNode().getNodeName().equals("UML:CompositeState.subvertex")) |
20         (currentNode.getNodeName().equals("UML:CompositeState") &
21          currentNode.getParentNode().getNodeName().equals("UML:Transition.source")) |
22         (currentNode.getNodeName().equals("UML:CompositeState") &
23          currentNode.getParentNode().getNodeName().equals("UML:Transition.target")) |
24         currentNode.getNodeName().equals("UML:Pseudostate") |
25         currentNode.getNodeName().equals("UML:Class")) {
26             NamedNodeMap attributeNodes = currentNode.getAttributes();
27             for ( int i = attributeNodes.getLength()-1; i >=0; i--){
28               Attr attribute = ( Attr ) attributeNodes.item( i );
29               if (attribute.getNodeName().equals("name") &
30                 attribute.getOwnerElement().getNodeName().equals("UML:SimpleState")) {
31                       state_name = attribute.getNodeValue();
32               }
33               if (attribute.getNodeName().equals("xmi.id") &
34                 attribute.getOwnerElement().getNodeName().equals("UML:SimpleState")) {
35                       id_to_name.put(attribute.getNodeValue(), state_name);
36                       all_states.add(attribute.getNodeValue());
37                       if (!encl_state_id.equals("")) {
38                             childparent_put(attribute.getNodeValue(), encl_state_id);
39                         }
40               }
```

Figure B.2: Statechart element processing code

of the various notational elements. The key factor to consider when choosing a mapping implementation is time complexity. In our implementation, we use a map as the time taken to determine whether the mapping contains a corresponding name for a particular identifier is constant.

2. The core data structure of the tree traverser is a table which takes the form of a 2-dimensional array of 5 columns. Each transition recovered from the XMI representation is stored in the table. The 5 columns of the table hold the identifiers of source state, event, guard-condition, action and target state, respectively. The time for locating a transition in the table is proportional to the number of transitions in a UML statechart diagram.

3. To overcome the name conflict problem of notational elements, the identifier of an XML element is used as an unique identifier for the corresponding notational element. The adoption of this approach reduces the complexity of the implementation by not manipulating another set of internally defined identifiers.

By way of commentary on the translation code in Figures B.2, B.3, B.4, B.5, B.6 and B.7, we make the following observations:

lines 5–8: If the received node is a document node, the method *getChildNodes* is invoked (line 7) to obtain a list of all child nodes of the received node. The method *processChildNodes* (Figures B.6 and B.7) then processes each of the child nodes by executing the method *processNode* as specified on lines 166, 198 and 202.

lines 27 and 61–69: The *for* statement on line 27 iterates over the list of attribute nodes of the current node. If the current node is an element node and is an event, the name and identifier of the event are retrieved from the attribute nodes using the method *getNodeValue* (lines 63 and 68). The name and identifier of the event are then stored in a set *events* and a map *id_to_name*.

lines 29–60 and 70–121: The names and identifiers of other notational elements which include state, guard-condition, action, transition and composite state are extracted using a similar approach. Lines 29–40, 41–50, 51–60, 70–109 and 110–121 detail, respectively, the extraction of state, guard-condition, action, transition and composite state. The identifiers of event trigger, non-composite source state, composite source state, non-composite target state and composite target state of transitions are retrieved by the method *getNodeValue* on lines 74, 80, 92, 98 and 107. The values of the variables *source_id*, *event_id*, *guard_id*, *action_id* and *target_id* are then assigned to a 2-dimensional array *transition* on lines 177–187 for facilitatng the generation of the π-calculus specification.

After collecting all the required information by the tree traverser, we now discuss how the π-calculus expressions are generated. The implementation considerations of the code generator are enumerated as follows:

1. The code generation process is divided into three steps. Firstly, the parameters of a process identifier are constructed and the process identifier corresponding to

```
41                if (attribute.getNodeName().equals("name") &
42                  attribute.getOwnerElement().getNodeName().equals("UML:Guard")) {
43                    guard_name = attribute.getNodeValue();
44                    guards.add(guard_name);
45                }
46                if (attribute.getNodeName().equals("xmi.id") &
47                  attribute.getOwnerElement().getNodeName().equals("UML:Guard")) {
48                  guard_id = attribute.getNodeValue();
49                  id_to_name.put(guard_id, guard_name);
50                }
51                if (attribute.getNodeName().equals("name") &
52                  attribute.getOwnerElement().getNodeName().equals("UML:CallAction")) {
53                    action_name = attribute.getNodeValue();
54                    actions.add(action_name);
55                }
56                if (attribute.getNodeName().equals("xmi.id") &
57                  attribute.getOwnerElement().getNodeName().equals("UML:CallAction")) {
58                        action_id = attribute.getNodeValue();
59                    id_to_name.put(action_id, action_name);
60                }
61                if (attribute.getNodeName().equals("name") &
62                  attribute.getOwnerElement().getNodeName().equals("UML:CallEvent")) {
63                    event_name = attribute.getNodeValue();
64                    events.add(event_name);
65                }
66                if (attribute.getNodeName().equals("xmi.id") &
67                  attribute.getOwnerElement().getNodeName().equals("UML:CallEvent")) {
68                    id_to_name.put(attribute.getNodeValue(), event_name);
69                }
70                if (attribute.getNodeName().equals("xmi.idref") &
71                  attribute.getOwnerElement().getNodeName().equals("UML:CallEvent") &
72                    attribute.getOwnerElement().getParentNode().getNodeName().
73                    equals("UML:Transition.trigger")) {
74                    event_id = attribute.getNodeValue();
75                }
76                if (attribute.getNodeName().equals("xmi.idref") &
77                  attribute.getOwnerElement().getNodeName().equals("UML:SimpleState") &
78                  attribute.getOwnerElement().getParentNode().getNodeName().
79                  equals("UML:Transition.source")) {
80                    source_id = attribute.getNodeValue();
81                    if (concurrent & !hasSuperState(source_id)) {
82                          if (class_name==null | class_name.equals(""))
83                            ch_table.put(source_id, "event_ch0");
84                          else
85                            ch_table.put(source_id, "event_" + class_name);
86                    }
87                }
88                if (attribute.getNodeName().equals("xmi.idref") &
89                  attribute.getOwnerElement().getNodeName().equals("UML:CompositeState") &
90                  attribute.getOwnerElement().getParentNode().getNodeName().
91                  equals("UML:Transition.source")) {
92                    source_id = attribute.getNodeValue();
93                }
```

Figure B.3: Statechart element processing code (continued)

the source state is then generated. Secondly, input action and matching construct representing the receipt of an event as well as π-calculus actions denoting guard-condition, action and interaction between a composite state and its substate(s) are produced. Finally, output action *step* and process identifier modelling the

```
94                if (attribute.getNodeName().equals("xmi.idref") &
95                  attribute.getOwnerElement().getNodeName().equals("UML:SimpleState") &
96                  attribute.getOwnerElement().getParentNode().getNodeName().
97                  equals("UML:Transition.target")) {
98                    target_id = attribute.getNodeValue();
99                    if (concurrent & !source_id.equals(target_id)) {
100                         ch_table.put(target_id, ch_table.get(source_id));
101                   }
102                 }
103               if (attribute.getNodeName().equals("xmi.idref") &
104                 attribute.getOwnerElement().getNodeName().equals("UML:CompositeState") &
105                 attribute.getOwnerElement().getParentNode().getNodeName().
106                 equals("UML:Transition.target")) {
107                       target_id = attribute.getNodeValue();
108                       concur_state_no_of_regions_put(target_id, new Integer(no_of_orth_regions));
109                 }
110               if (attribute.getNodeName().equals("name") &
111                 attribute.getOwnerElement().getNodeName().equals("UML:CompositeState")) {
112                       encl_state_name = attribute.getNodeValue();
113               }
114               if (attribute.getNodeName().equals("xmi.id") &
115                 attribute.getOwnerElement().getNodeName().equals("UML:CompositeState")) {
116                   encl_state_id = attribute.getNodeValue();
117                   id_to_name.put(encl_state_id, encl_state_name);
118                   ch = "event_ch" + no_of_orth_regions;
119                   ch_table.put(encl_state_id, ch);
120                   all_states.add(attribute.getNodeValue());
121               }
122               if (attribute.getNodeName().equals("isConcurrent") &
123                 attribute.getOwnerElement().getNodeName().equals("UML:CompositeState") &
124                 attribute.getNodeValue().equals("true")) {
125                   concurrent = true;
126                   no_of_orth_regions = 0;
127               }
```

Figure B.4: Statechart element processing code (continued)

target state are yielded.

2. The generation of π-calculus specifications is a non-trivial process as:

   (i) The existence of non-concurrent composite states, concurrent composite states and interlevel transitions complicates the development as well as the testing of the code generator.

   (ii) In the π-calculus representations, substates which belong to the same orthogonal region of a concurrent composite state should use the same channel for communicating with the concurrent composite state, while substates which are in different orthogonal regions should use different channels for interacting with the concurrent composite state. To ensure that the correct channel is used, a table is maintained for keeping track of the channels for the substates of a concurrent composite state.

```
128                if (attribute.getNodeName().equals("name") &
129                  attribute.getOwnerElement().getNodeName().equals("UML:SimpleState") &
130                  attribute.getOwnerElement().getParentNode().getNodeName().
131                    equals("UML:CompositeState.subvertex") &
132                  attribute.getOwnerElement().getParentNode().getParentNode().getNodeName().
133                    equals("UML:CompositeState")) {
134                      NamedNodeMap comp_state_attributeNodes =
135                        attribute.getOwnerElement().getParentNode().getParentNode().
136                        getAttributes();
137                      for ( int k = comp_state_attributeNodes.getLength()-1; k >=0; k--){
138                        Attr comp_state_attribute = (Attr) comp_state_attributeNodes.item(k);
139                        if (comp_state_attribute.getNodeName().equals("isConcurrent") &
140                            comp_state_attribute.getNodeValue().equals("true")) {
141                              concur_substates.add(attribute.getNodeValue());
142                        }
143                      }
144                }
145                ... /* process other element nodes */
146              }
147            }
148         processChildNodes( currentNode.getChildNodes() );
149         break;
150     }
151   }
```

Figure B.5: Statechart element processing code (continued)

```
152   public void processChildNodes( NodeList children )
153   {
154     if ( children.getLength() != 0 )
155         for ( int i = 0; i < children.getLength(); i++){
156           if (children.item(i).getNodeName().equals("UML:Transition") &&
157             children.item(i).getParentNode().getNodeName().
158             equals("UML:StateMachine.transitions")) {
159              source_id = "";
160              event_id = "";
161              guard_id = "";
162              action_id = "";
163              target_id = "";
164              guard_exp_id = "";
165              action_exp_id = "";
166              processNode( children.item( i ) );
167              if (hasSubStates(source_id)) {
168                ...  /* process transitions of composite state */
169              }
170              else
171                if (hasSuperState(source_id) &&
172                    !(encl_state(target_id).equals(encl_state(source_id)))) {
173                       ...  /* process interlevel transition */
174                }
175                else
```

Figure B.6: Statechart element processing code (continued)

(iii) An event trigger of a join pseudostate is associated with the outgoing transition instead of the incoming transitions. As a result, the construction of a matching construct which involves a join pseudostate should refer to the

```
176                 if (!pseudostate_ids.contains(source_id)) {
177                       transition[no_of_trans][0] = source_id;
178                       transition[no_of_trans][1] = event_id;
179                       if (guard_exp_id.equals(""))
180                           transition[no_of_trans][2] = guard_id;
181                       else
182                           transition[no_of_trans][2] = guard_exp_id;
183                       if (action_exp_id.equals(""))
184                           transition[no_of_trans][3] = action_id;
185                       else
186                           transition[no_of_trans][3] = action_exp_id;
187                       transition[no_of_trans][4] = target_id;
188                       no_of_trans++;
189                 }
190                 else
191                   ...  /* process transition of the pseudostate */
192           }
193           else
194             if (children.item(i).getNodeName().equals("UML:CompositeState") &&
195                 children.item(i).getParentNode().getNodeName().
196                 equals("UML:CompositeState.subvertex")) {
197                   ...                              /* process composite state  */
198                   processNode( children.item( i ) );
199                   ...
200             }
201             else
202                   processNode( children.item( i ) );
203         }
204   }
```

Figure B.7: Statechart element processing code (continued)

event trigger of the outgoing transition.

(iv) A transition in which the target state is a join pseudostate should use the target state of the outgoing transition of the join pseudostate for generating the π-calculus specification.

According to Rules 1 and 2 of the formalization in Chapter 3, each event which is represented as a channel is a parameter of the process identifier. An iterator is used for retrieving the event names from the set *events* collected by the tree traverser. The generated event parameters are stored in a variable *e_params*. Lines 1–8 of Figure B.8 are the code for an iterator that processes the set *events*.

Likewise, the code fragments for guard-conditions and actions are on lines 9–12 and 13–37 of Figure B.8, respectively. Channels related to guard-conditions and actions are stored in variables *g_params* and *a_params*. All these variables and state name are then concatenated and stored in a variable *source_state_exp*.

Consider a UML statechart diagram that comprises three states S_1, S_2 and S_3 as well as two event triggers e_1 and e_2 as illustrated in Figure B.9. The XMI repre-

```
1  Iterator eit = events.iterator();
2  while (eit.hasNext()) {
3      tmp_evt = (String) eit.next();
4      if (!processed_evts_params.contains(tmp_evt)) {
5          e_params = e_params + "," + tmp_evt;
6          processed_evts_params.add(tmp_evt);
7      }
8  }
9  Iterator git = guards.iterator();
10 while (git.hasNext()) {
11     g_params = g_params + "," + (String)git.next();
12 }
13 Iterator ait = actions.iterator();
14 while (ait.hasNext()) {
15     tmp_act = (String)ait.next();
16     if (tmp_act.substring(0,4).equals("send")) {
17       startpos = 5;
18       endpos = tmp_act.indexOf('.');
19       rcvd_obj = tmp_act.substring(startpos,endpos);
20       if (!processed_params.contains("ins_" + rcvd_obj)) {
21           a_params = a_params + "," + "ins_" + rcvd_obj;
22           processed_params.add("ins_" + rcvd_obj);
23       }
24       startpos = endpos + 1;
25       endpos = tmp_act.length();
26       tmp_evt = tmp_act.substring(startpos,endpos);
27       if (!processed_params.contains(tmp_evt)) {
28           a_params = a_params + "," + tmp_evt;
29           processed_params.add(tmp_evt);
30       }
31     }
32     else
33       if (!processed_params.contains(tmp_act)) {
34           a_params = a_params + "," + tmp_act;
35           processed_params.add(tmp_act);
36       }
37 }
```

Figure B.8: Constructing parameters for process identifier

sentation of the UML statechart diagram generated by Poseidon for UML is shown in Figures B.10 and B.11. For the sake of clarity, unnecessary detail is elided and marked by an ellipsis. Lines 1 and 2 identify that XML version 1.0, UTF-8 encoding and XMI version 1.2 are used. Lines 4–16, 17–32 and 33–45 are the XMI representations of the states S_1, S_2 and S_3, respectively. Likewise, the XMI representations of the two transitions and two event triggers correspond to lines 48–66, 67–85, 88–97 and 98–107.

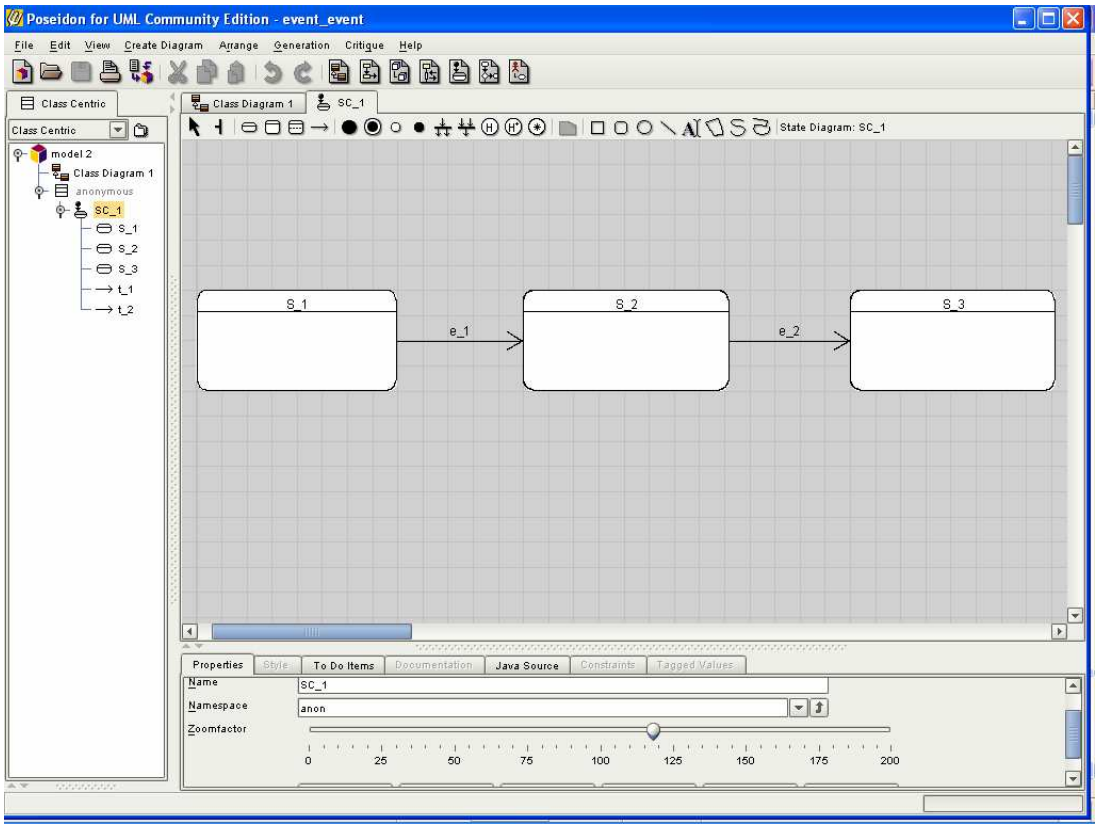

Figure B.9: The screenshot of example 1

Figure B.12 shows the MWB code (π-calculus representation) of Figure B.9 generated by the SC2PiCal. Line 1 of Figure B.12 is based on the variable *source_state_exp*. To generate the input action on line 2 of Figure B.12, we define two variables as:

```
1  String  input_ch = "x";
2  String  ich;
```

Figure B.13 is a block of code for building an input action. We consider the following cases:

```
1  <?xml version = '1.0' encoding = 'UTF-8' ?>
2  <XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Fri Nov 18 23:43:08 CST 2005'>
3  ...
4  <UML:SimpleState xmi.id = 'a14' name = 'S_1' isSpecification = 'false'>
5    <UML:ModelElement.taggedValue>
6      <UML:TaggedValue xmi.id = 'a15' isSpecification = 'false'
7                       dataValue = '-64--88-1-100-779dce:107a3fee6e8:-7ff7'>
8        <UML:TaggedValue.type>
9          <UML:TagDefinition xmi.idref = 'a3'/>
10       </UML:TaggedValue.type>
11     </UML:TaggedValue>
12   </UML:ModelElement.taggedValue>
13   <UML:StateVertex.outgoing>
14     <UML:Transition xmi.idref = 'a16'/>
15   </UML:StateVertex.outgoing>
16 </UML:SimpleState>
17 <UML:SimpleState xmi.id = 'a17' name = 'S_2' isSpecification = 'false'>
18   <UML:ModelElement.taggedValue>
19     <UML:TaggedValue xmi.id = 'a18' isSpecification = 'false'
20                      dataValue = '-64--88-1-100-779dce:107a3fee6e8:-7ff6'>
21       <UML:TaggedValue.type>
22         <UML:TagDefinition xmi.idref = 'a3'/>
23       </UML:TaggedValue.type>
24     </UML:TaggedValue>
25   </UML:ModelElement.taggedValue>
26   <UML:StateVertex.outgoing>
27     <UML:Transition xmi.idref = 'a19'/>
28   </UML:StateVertex.outgoing>
29   <UML:StateVertex.incoming>
30     <UML:Transition xmi.idref = 'a16'/>
31   </UML:StateVertex.incoming>
32 </UML:SimpleState>
33 <UML:SimpleState xmi.id = 'a20' name = 'S_3' isSpecification = 'false'>
34   <UML:ModelElement.taggedValue>
35     <UML:TaggedValue xmi.id = 'a21' isSpecification = 'false'
36                      dataValue = '-64--88-1-100-779dce:107a3fee6e8:-7ff5'>
37       <UML:TaggedValue.type>
38         <UML:TagDefinition xmi.idref = 'a3'/>
39       </UML:TaggedValue.type>
40     </UML:TaggedValue>
41   </UML:ModelElement.taggedValue>
42   <UML:StateVertex.incoming>
43     <UML:Transition xmi.idref = 'a19'/>
44   </UML:StateVertex.incoming>
45 </UML:SimpleState>
46 ...
```

Figure B.10: The XMI representation of example 1

1. If both the source and target states are substates of a concurrent composite state, the input action is generated by the statements on lines 6–8.

2. If the source state is a substate of a concurrent composite state and the target state is a basic state in which the transition is an interlevel transition, the input action is produced by the statements on lines 12–14.

3. If the source state is a substate of a non-concurrent composite state and a concurrent composite state is present, the input action is yielded by the statement on line 17.

4. If the source state is a substate of a non-concurrent composite state and a con-

```
47  <UML:StateMachine.transitions>
48    <UML:Transition xmi.id = 'a16' name = 't_1' isSpecification = 'false'>
49      <UML:ModelElement.taggedValue>
50        <UML:TaggedValue xmi.id = 'a22' isSpecification = 'false'
51                         dataValue = '-64--88-1-100-779dce:107a3fee6e8:-7ff4'>
52          <UML:TaggedValue.type>
53            <UML:TagDefinition xmi.idref = 'a3'/>
54          </UML:TaggedValue.type>
55        </UML:TaggedValue>
56      </UML:ModelElement.taggedValue>
57      <UML:Transition.trigger>
58        <UML:CallEvent xmi.idref = 'a23'/>
59      </UML:Transition.trigger>
60      <UML:Transition.source>
61        <UML:SimpleState xmi.idref = 'a14'/>
62      </UML:Transition.source>
63      <UML:Transition.target>
64        <UML:SimpleState xmi.idref = 'a17'/>
65      </UML:Transition.target>
66    </UML:Transition>
67    <UML:Transition xmi.id = 'a19' name = 't_2' isSpecification = 'false'>
68      <UML:ModelElement.taggedValue>
69        <UML:TaggedValue xmi.id = 'a24' isSpecification = 'false'
70                         dataValue = '-64--88-1-100-779dce:107a3fee6e8:-7ff3'>
71          <UML:TaggedValue.type>
72            <UML:TagDefinition xmi.idref = 'a3'/>
73          </UML:TaggedValue.type>
74        </UML:TaggedValue>
75      </UML:ModelElement.taggedValue>
76      <UML:Transition.trigger>
77        <UML:CallEvent xmi.idref = 'a25'/>
78      </UML:Transition.trigger>
79      <UML:Transition.source>
80        <UML:SimpleState xmi.idref = 'a17'/>
81      </UML:Transition.source>
82      <UML:Transition.target>
83        <UML:SimpleState xmi.idref = 'a20'/>
84      </UML:Transition.target>
85    </UML:Transition>
86  </UML:StateMachine.transitions>
87  ...
88  <UML:CallEvent xmi.id = 'a23' name = 'e_1' isSpecification = 'false'>
89    <UML:ModelElement.taggedValue>
90      <UML:TaggedValue xmi.id = 'a26' isSpecification = 'false'
91                       dataValue = '-64--88-1-100-779dce:107a3fee6e8:-7ff2'>
92        <UML:TaggedValue.type>
93          <UML:TagDefinition xmi.idref = 'a3'/>
94        </UML:TaggedValue.type>
95      </UML:TaggedValue>
96    </UML:ModelElement.taggedValue>
97  </UML:CallEvent>
98  <UML:CallEvent xmi.id = 'a25' name = 'e_2' isSpecification = 'false'>
99    <UML:ModelElement.taggedValue>
100     <UML:TaggedValue xmi.id = 'a27' isSpecification = 'false'
101                      dataValue = '-64--88-1-100-779dce:107a3fee6e8:-7ff1'>
102       <UML:TaggedValue.type>
103         <UML:TagDefinition xmi.idref = 'a3'/>
104       </UML:TaggedValue.type>
105     </UML:TaggedValue>
106   </UML:ModelElement.taggedValue>
107 </UML:CallEvent>
108 ...
109 </XMI>
```

Figure B.11: The XMI representation of example 1 (continued)

current composite state is absent, the input action is emitted by the statement on line 22.

5. If the source state is a concurrent composite state, the input action is obtained

```
1  agent S_1(step,event_ch0,e_1,e_2)= \
2  event_ch0(x1). \
3  ([x1=e_1] \
4  'step.S_2(step,event_ch0,e_1,e_2)+ \
5  [x1=e_2]'step.S_1(step,event_ch0,e_1,e_2))
6
7  agent S_2(step,event_ch0,e_1,e_2)= \
8  event_ch0(x1). \
9  ([x1=e_2] \
10 'step.S_3(step,event_ch0,e_1,e_2)+ \
11 [x1=e_1]'step.S_2(step,event_ch0,e_1,e_2))
```

Figure B.12: The MWB code of example 1

by running the statement on line 28. The *for* loop on lines 29–48 is to output actions which represent the interaction between the concurrent composite state and its substates.

6. If the source state is not a concurrent composite state and a concurrent composite state is present, the input action is determined by the statement on line 53.

7. If the source state is not a concurrent composite state and a concurrent composite state is absent, the input action is constructed by the statement on line 57.

Line 2 of Figure B.12 is obtained by executing statements on lines 1, 57 and 60 of Figure B.13.

Figure B.14 is a snippet of code which produces matching construct. The cases whether an event is present or not in a transition are treated separately. The presence of an event in a transition is handled by line 2 of Figure B.14, while the absence of an event in a transition is processed by either lines 7–9 or line 12. If the source state is a substate of a concurrent composite state and the target state is a join pseudostate, the event of the outgoing transition of the join pseudostate is used to build the matching construct using the statements on lines 7 and 8 of Figure B.14 as the incoming transitions of the join pseudostate are triggerless transitions. The matching construct on line 3 of Figure B.12 is constructed using line 2 of Figure B.14.

The code fragment which generates code for target state is shown in Figures B.15 and B.16. In what follows, we summarize some of the most important cases considered when constructing code for target state.

lines 3–8: are executed if the target state has a superstate and the superstates of the

```
ich = input_ch + i;
if (hasSuperState(transition[index][0])) {
    if (concurrent) {
        if (concur_substates.contains(id_to_name.get(transition[index][0])) &
            concur_substates.contains(id_to_name.get(transition[index][4]))) {
           ch_no = ((String) concur_ch_table.get(transition[index][0])).substring(14);
           input_exp = concur_ch_table.get(transition[index][0]) +
                       "(" + ich + ",ack_substate" + ch_no + ")";
        }
        else {
           if (concur_substates.contains(id_to_name.get(transition[index][0]))) {
             ch_no = ((String) concur_ch_table.get(tmp_state)).substring(14);
             input_exp = (String) concur_ch_table.get(tmp_state) + "(" + ich + ",ack_substate" +
                         ch_no + ")";
           }
           else {
             input_exp = ch_table.get(transition[index][0]) + "(" + ich + ",ack" + ")";
           }
        }
    }
    else {
        input_exp = event_ch1 + "(" + ich + ",ack" + ")";
    }
}
else {
    if (concurrent) {
        if (concur_comp_states.contains(id_to_name.get(transition[index][0]))) {
          input_exp = ch_table.get(transition[index][0]) + "(" + input_ch + "1" + ")" + ".";
          for (m=1; m<=((Integer)comp_state_no_of_orth_regions_table.get(
                 id_to_name.get(transition[index][0]))).intValue(); m++) {
            ack_ch_no = m + 1;
            if (m==1) {
                channel_list = "ack_substate" + m;
                send_event_list = "'event_substate" + m +
                                  "<" + input_ch + "1" + "," + "ack_substate" + m + ">";
                recd_ack_list = "ack_substate" + m +
                                "(" + input_ch + ack_ch_no + ")";
            }
            else {
                channel_list = channel_list + "," + "ack_substate" + m;
                send_event_list = send_event_list + "." +
                                  "'event_substate" + m +
                                  "<" + input_ch + "1" + "," + "ack_substate" + m + ">";
                recd_ack_list = recd_ack_list + "." +
                                "ack_substate" + m +
                                "(" + input_ch + ack_ch_no + ")";
            }
          }
          input_exp = input_exp + "(" + "^" + channel_list + ")" + " \\" + "\n" +
                      send_event_list + "." + " \\" + "\n" + recd_ack_list;
        }
        else {
          input_exp = ch_table.get(transition[index][0]) + "(" + ich + ")";
        }
    }
    else {
        input_exp = event_ch0 + "(" + ich + ")";
    }
}
writeln(input_exp + "." + " \\");
```

Figure B.13: Generating input action

```
1  if (id_to_name.get(transition[index][1]) != null) {
2      e_match_exp = "[" + ich + "=" + id_to_name.get(transition[index][1]) + "]";
3  }
4  else
5    if (concur_substates.contains(id_to_name.get(transition[index][0])) &
6        joins.contains(transition[index][4])) {
7      join_event = locate_join_event(transition[index][4]);
8      e_match_exp = "[" + ich + "=" + join_event + "]";
9      processed_events.add(join_event);
10   }
11   else {
12     e_match_exp = "";
13   }
```

Figure B.14: Generating matching construct

source and target states are different.

lines 16–34: are run whenever the target state is a concurrent composite state.

lines 38–39: are invoked when the target state is a non-concurrent composite state.

lines 46–53: are executed only if both the source and target states are substates of a concurrent composite state.

lines 71–73: are invoked if the target state is a fork pseudostate.

lines 77–85: are evaluated whenever the target state is a join pseudostate.

lines 88–91: are run when the target state is a basic state.

The execution of lines 88–91 of Figure B.16 produces as its results a value for the variable *target_state_exp*. The output action *step* and target state of Figure B.12 are created by concatenating the output action *step* with the variable *target_state_exp*.

The operator + and continuation character \ on line 4 of Figure B.12 as well as the matching construct for other event i.e. line 5 of Figure B.12 are generated by the code fragment as shown in Figure B.17.

The operator + and continuation character \ are produced by the method *writeln* on line 7. The *if* statement on lines 5 and 6 specify that any events which are not equal to event *e_1* (transition[index][1]) and have not been processed are regarded as other events. A π-calculus expression which consists of a matching construct, an output prefix *step* and a process identifier is generated for each of these events.

```
if (hasSuperState(transition[index][4]) &&
    !(encl_state(transition[index][4]).equals(encl_state(transition[index][0])))) {
        target_params = event_ch1 + e_params + g_params + a_params + other_params;
        target_state_exp = id_to_name.get(transition[index][4]) + "(" + target_params + ")";
        target_state_exp = "(" + "^" + event_ch1 + ")" + " \\" + "\n" +
                           "(" + id_to_name.get(encl_state(transition[index][4])) +
                           "(" + "step," + event_ch0 + "," + target_params + ")" + " | " +
                           " \\" + "\n" + target_state_exp + ")";
}
else
    if (transition[index][4].equals("0"))
      target_state_exp = "0";
    else {
      if (!(concur_state.indexOf(transition[index][4])==-1) &
          (transition[index][1]!=null)) {
          ListIterator lt = comp_state.listIterator();
          while (lt.hasNext()) {
              if (lt.next().equals(transition[index][4])) {
                  target_params = ch_table.get((String) default_state.get(l)) +
                                  e_params + g_params + a_params + other_params;
                  ch_list = ch_list + "," + ch_table.get((String) default_state.get(l));
                  restrict_list = restrict_list + "(" + "^" +
                                  ch_table.get((String) default_state.get(l)) + ")";
                  target_state_exp = target_state_exp + " | " + " \\" + "\n" +
                                     id_to_name.get((String) default_state.get(l)) +
                                     "(" + target_params + ")";
              }
              l++;
          }
          target_state_exp = restrict_list +
                             "(" + id_to_name.get(transition[index][4]) +
                             "(" + "step" + "," + event_ch0 + e_params + g_params +
                             a_params + other_params + ch_list + ")" +
                             target_state_exp + ")";
      }
      else
        if (hasSubStates(transition[index][4])) {
          target_state_exp = id_to_name.get(transition[index][4]) +
                             "(" + params + ")";
        }
        else {
          if (hasSuperState(transition[index][4])) {
            if (concurrent) {
              if (concur_substates.contains(id_to_name.get(transition[index][0])) &
                  concur_substates.contains(id_to_name.get(transition[index][4]))) {
                ch_no = ((String) concur_ch_table.get(transition[index][0])).substring(14);
                target_params = concur_ch_table.get(transition[index][0]) +
                                e_params + g_params + a_params + other_params + "," +
                                "cont_substate" + ch_no + "," +
                                "end_substate" + ch_no;
                target_state_exp = "cont_substate" + ch_no + "." +
                                   id_to_name.get(transition[index][4]) + "(" +
                                   target_params + ")";
              }
              else {
                target_params = ch_table.get(transition[index][4]) + e_params + g_params +
                                a_params + other_params;
                target_state_exp = id_to_name.get(transition[index][4]) + "(" +
                                   target_params + ")";
              }
            }
            else {
              target_params = event_ch1 + e_params + g_params + a_params +
                              other_params;
              target_state_exp = id_to_name.get(transition[index][4]) + "(" +
                                 target_params + ")";
            }
          }
          else {
```

Figure B.15: Generating code for target state

```
70              if (forks.contains(transition[index][4])) {
71                 target_params = e_params + g_params + a_params + other_params;
72                 target_state_exp = gen_targets_for_a_fork(transition[index][4], target_params,
73                                                           event_ch0);
74              }
75              else {
76                if (joins.contains(transition[index][4])) {
77                  if (concur_substates.contains(id_to_name.get(transition[index][0]))) {
78                    target_state_exp = source_state_exp;
79                  }
80                  else {
81                    target_params = "step" + "," + event_ch0 + e_params + g_params + a_params +
82                                    other_params;
83                    target_state_exp = gen_targets_for_a_join(transition[index][4],target_params,
84                                                              event_ch0);
85                  }
86                }
87                else {
88                  target_params = "step" + "," + event_ch0 + e_params + g_params + a_params +
89                                  other_params;
90                  target_state_exp = id_to_name.get(transition[index][4]) + "(" +
91                                     target_params + ")";
92                }
93              }
94            }
95          }
96      }
```

Figure B.16: Generating code for target state (continued)

Based on the same principles, concepts and techniques, the other π-calculus expressions which correspond to guard-condition, action and composite state can also be generated as illustrated in the subsequent sections.

B.3 The Detailed Implementation of PiCal2NuSMV

The translation of UML statecharts based π-calculus into NuSMV code comprises three main steps as illustrated in Figure B.1. The lexical analysis and syntax analysis are done in the first two steps. A π-calculus specification is taken as input and the outputs are a number of lists and sets which contain state variable declaration, substate variable declaration and transition relation represented as a set of next statements. Subsequently, the code generator yields the corresponding NuSMV code.

ANTLR (ANother Tool for Language Recognition) is a software tool for generating a lexer and parser based on a specified grammar. To generate a lexer and parser for the π-calculus, we start from the widely-accepted BNF grammar for the π-calculus given in Figure B.18.

However, while suitable for typesetting the π-calculus, it is not appropriate for a

```
1  if (!e_match_exp.equals("")) {
2    Iterator eit2 = events.iterator();
3    while (eit2.hasNext()) {
4      String tmp_event = (String)eit2.next();
5      if (!tmp_event.equals(id_to_name.get(transition[index][1])) &&
6          !(processed_events.contains(tmp_event))) {
7        writeln("+" + " \\");
8        e_match_exp = "[" + ich + "=" + tmp_event + "]";
9        if (hasSuperState(transition[index][0]))
10         if (concur_substates.contains(id_to_name.get(transition[index][0])) &
11             concur_substates.contains(id_to_name.get(transition[index][4]))) {
12           ch_no = ((String) concur_ch_table.get(transition[index][0])).
13                    substring(14);
14           write(e_match_exp + "'ack_substate" + ch_no + "<neg>." +
15                 "cont_substate" + ch_no + "." + source_state_exp);
16         }
17         else {
18           if (concur_substates.contains(id_to_name.get(transition[index][0])) &
19               joins.contains(transition[index][4])) {
20             ch_no = ((String) concur_ch_table.get(tmp_state)).substring(14);
21             write(e_match_exp + "'ack_substate" + ch_no + "<neg>." +
22                   "cont_substate" + ch_no + "." + source_state_exp);
23           }
24           else {
25             write(e_match_exp + "'ack<neg>." +
26                   source_state_exp);
27           }
28         }
29       else {
30         write(e_match_exp + "'step." +
31               source_state_exp);
32       }
33     }
34   }
35   writeln(")");
36 }
37 else {
38   writeln("");
39 }
```

Figure B.17: Generating matching constructs for other events

$$P ::= \mathbf{0} \mid x(\vec{y}).P \mid \overline{x}\langle\vec{y}\rangle.P \mid (\boldsymbol{\nu}\vec{x})P \mid [x = y]P \mid P|P \mid P + P \mid A(\vec{x})$$

Figure B.18: The grammar for the π-calculus

machine processable representation. Consequently, we use the EBNF grammar corresponding to the input language for the Mobility Workbench. A fragment of the ANTLR input specification which includes both the EBNF grammar and actions are given in Figures B.19 and B.20.

```
1  class PiCalParser extends Parser;
2  options {k = 2;}
3
4  multiDefn    : (agentDefn)* EOF;
5  agentDefn    :   {String v_agentId;}
6                  AGENT v_agentId=agentId
7                    {PiCal2NuSMV.set_cur_state(v_agentId);
8                     PiCal2NuSMV.set_init_state(v_agentId);}
9                  EQ process
10                   {PiCal2NuSMV.reset_superstate_enable_flag();};
11 agentId            returns [String value = new String()]
12               :   {String v_channelList;}
13                 v_agentname:AGENTNAME
14                   {PiCal2NuSMV.set_cur_agent_name(v_agentname.getText());}
15                 LPAREN v_channelList=channelList RPAREN
16                   {value = v_agentname.getText();
17                    if (PiCal2NuSMV.get_gen_substate_flag())
18                      PiCal2NuSMV.append_substate_var_decl(value);
19                    else
20                      PiCal2NuSMV.append_state_var_decl(value);};
```

Figure B.19: Fragment of the ANTLR input specification

Line 1 is a parser declaration which specifies the parser to be generated. Line 2 is the options section. The option k=2 defines a token lookahead value of two. Line 4 is a parser rule. The rule states that a valid π-calculus file contains multiple agent definitions followed by an end-of-file token. Lines 5–10 stipulate the second parser rule. The statements on lines 5, 7, 8 and 10, which are embedded in the EBNF grammar and enclosed in braces, are actions. These embedded actions are incorporated into the source code of the parser. They store information related to state variable declaration, substate variable declaration and transition relation to a number of lists and sets when expressions are parsed. Likewise, the parser rules for agent identifiers, non-deterministic choices, parallel compositions, matching constructs and restrictions are defined on lines

```
21  channelList     returns [String value = new String()]
22                : v_channel:CHANNEL
23                  {value = v_channel.getText();
24                   if (!PiCal2NuSMV.retrieve_cur_agent_name().equals("")) {
25                     PiCal2NuSMV.linkup_substate_channels(
26                       PiCal2NuSMV.retrieve_cur_agent_name(),value);
27                   }
28                   PiCal2NuSMV.clear_cur_agent_name();
29                  }
30                  (COMMA CHANNEL)*;
31  process       : sumExpr;
32  sumExpr       : parallelExpr
33                  ({if (PiCal2NuSMV.is_superstate_enable())
34                      PiCal2NuSMV.append_tmp_case_cond(PiCal2NuSMV.get_common_cond());}
35                   PLUS parallelExpr)*
36                   {PiCal2NuSMV.reset_ack_substate_flag();};
37  parallelExpr : expr (PARALLEL
38                  {PiCal2NuSMV.set_gen_substate_flag();
39                   PiCal2NuSMV.chk_concur_comp_state();
40                   PiCal2NuSMV.append_multi_list("substate");
41                   PiCal2NuSMV.rm_frm_tmp_var_list("state");
42                  } expr)*
43                  {PiCal2NuSMV.reset_vars();
44                   PiCal2NuSMV.reset_gen_substate_flag();};
45  expr          : (match)* subExpr;
46  subExpr       : (restrict)? term;
47  ...
```

Figure B.20: Fragment of the ANTLR input specification (continued)

11–20, 32–36, 37–44, 45 and 46, respectively.

Next, ANTLR constructs a lexer and a parser from the specification in Figures B.19 and B.20, instances of which may be created thus:

```
1  try {
2    input = new DataInputStream(new FileInputStream(fname));
3    module_name = fname;
4  } catch(Exception e) {System.err.println(e.getMessage());}
5  PiCalLexer lexer = new PiCalLexer(input);
6  PiCalParser parser = new PiCalParser(lexer);
7  gen_module();
```

A *DataInputStream* object based on the filename of the π-calculus expressions is created. The filename of the π-calculus expressions is also used as the module name. A lexer and parser are created, respectively, by using the *DataInputStream* object and lexer as arguments of the constructors.

Once a parser has been constructed, we can address the transformation of the π-calculus representation into NuSMV code for the purpose of model-checking. Implementation issues and considerations of the translation module include:

1. Like the code generator of SC2PiCal, non-concurrent composite states, concurrent composite states and interlevel transitions cause the implementation to become more complex.

2. As defined by Rule 12 of Chapter 5, the signals *cont* and *end* in a π-calculus specification are represented in NuSMV as Boolean variables. The number of *cont* and *end* variables to be generated in the VAR section depends on the number of the orthogonal regions of the concurrent composite state. The number of orthogonal regions is obtained indirectly from a π-calculus process definition by counting the number of concurrent processes in the process definition.

3. The key data structure is a set *var_list* which keeps a set of NuSMV variables to be included in the ASSIGN section. The ASSIGN section is constructed by looping over the set *var_list* and retrieving all the corresponding next statements from the list *case_cond_stmt*.

The starting point is the method *gen_module*, the implementation of which is given below:

```
1 public static void gen_module() {
2   gen_module_decl();
3   gen_var_decl();
4   gen_assign_stmts();
5   gen_fairness_stmt();
6 }
```

The structure of the method *gen_module* coincides with the structure of a NuSMV module:

line 2: the module declaration,

line 3: the VAR section,

line 4: the ASSIGN section and

line 5: the fairness constraint.

The fairness constraint specifies that the module is selected for execution infinitely often. The following code fragments implement the method *gen_module_decl* and its two related methods *get_module_name* and *to_comma_sep_list*:

```
1  private static void gen_module_decl() {
2    writeln("MODULE " + get_module_name(module_name) +
3            "(" + to_comma_sep_list(param_list) + ")");
4  }
5
6  private static String get_module_name(String in_string) {
7     int startPos = in_string.indexOf('\\')+ 1;
8     int endPos = in_string.indexOf('.');
9     String module_name = in_string.substring(startPos, endPos);
10    return module_name;
11 }
12
13 private static String to_comma_sep_list(Set in_set) {
14   String out_string = "";
15   Iterator it = in_set.iterator();
16   while (it.hasNext()) {
17     if (out_string.equals(""))
18       out_string += (String) it.next();
19     else
20       out_string += (String) "," + it.next();
21   }
22   return out_string;
23 }
```

The method *gen_module_decl* outputs the module declaration (line 1 of Figure B.21) to a file. The method *get_module_name* inputs a string which consists of a subdirectory, if any, a filename as well as an extension and returns the filename as the module name. The method *to_comma_sep_list* returns all the elements of the set *param_list* as a string in which each element is separated by a comma.

The method *gen_var_decl*, which outputs variable declarations, is implemented as Figures B.22 and B.23. Lines 2 and 3 of Figure B.21 are generated by lines 3–18 of the method *gen_var_decl*. The symbolic values of the scalar variable *state* are retrieved from the set *state_var_decl*.

By implementing the methods *gen_assign_stmts* and *gen_fairness_stmt* using a similar strategy, the ASSIGN section (lines 4–24 of Figure B.21) and the fairness constraint (line 25 of Figure B.21) are generated. The method *gen_assign_stmts* (Figure B.24) produces line 4 of Figure B.21 and invokes the method *retrieve_case_stmt* (Figures B.25 and B.26) for constructing the initial and next statements. The set *var_list* (line 5 of Figure B.25) stores the set of NuSMV variables in which the transition relation is defined in the ASSIGN section. Similarly, each element of the lists *case_cond_varname* (line 8 of Figure B.25) and *case_cond_stmt* (line 13 of Figure B.25) holds a NuSMV variable name and its corresponding next statement which consists of the precondition and the next possible value(s) of the NuSMV variable. For each NuSMV variable in the

```
1  MODULE sc_1(step,event_buff)
2  VAR
3  state: {S_3,S_1,S_2};
4  ASSIGN
5  init(state) := S_1;
6  next(state) :=
7  case
8  state=S_1 & event_buff=e_1 & !step:S_2;
9  state=S_2 & event_buff=e_2 & !step:S_3;
10 1: state;
11 esac;
12 next(step) :=
13 case
14 state=S_1 & event_buff=e_1 & !step:1;
15 state=S_2 & event_buff=e_2 & !step:1;
16 state=next(state) & !step: 1;
17 event_buff != empty & step: 0;
18 1: step;
19 esac;
20 next(event_buff):=
21 case
22 event_buff != empty & next(step): empty;
23 1 : event_buff;
24 esac;
25 FAIRNESS running
```

Figure B.21: The NuSMV code of example 1

set *var_list*, the list *case_cond_varname* is looped over. If the NuSMV variable exists in the list *case_cond_varname* and is a variable defined within the NuSMV module, an initial statement is generated. Additionally, all the associated next statements are fetched from the list *case_cond_stmt* and then emitted by the method *retrieve_case_stmt*.

Executing lines 17 and 18 of Figure B.25 as well as lines 29 and 30 of Figure B.25 output the *init* statement, *next* statement and *case* statement on lines 5–7 of Figure B.21. Lines 8 and 9 of Figure B.21 are constructed by executing line 32 of Figure B.25 and the *while* statement on line 10 of Figure B.25 twice. Lines 10 and 11 of Figure B.21 are generated by lines 78 and 81 of Figure B.26. Likewise, lines 12–15 of Figure B.21 as well as lines 18 and 19 of Figure B.21 are obtained. The statements on lines 16, 17 and 20–24 of Figure B.21 are produced by lines 39, 59 and 85 of Figure B.26. The method *gen_fairness_stmt*, which yields line 25 of Figure B.21, is defined below:

```
1  private static void gen_fairness_stmt() {
2    writeln("FAIRNESS running");
3  }
```

```
1  private static void gen_var_decl() {
2    int index;
3    writeln("VAR");
4    write("state: ");
5    write("{");
6    index = 1;
7    Iterator it;
8    it = state_var_decl.iterator();
9    while (it.hasNext()) {
10     if (index == 1) {
11       write(it.next());
12     }
13     else {
14       write("," + it.next());
15     }
16     index++;
17   }
18   writeln("};");
19   if (!substate_var_decl.isEmpty()) {
20     if (concur_comp_state_flag) {
21       for (int i=0; i<=no_of_regions; i++) {
22         if (substate[i].size() > 0) {
23           writeln("substate" + i + ": {" + to_comma_sep_list(substate[i]) +
24                   ",nil" + "};");
25           writeln("ack_substate" + i + ": {pos, neg, undefine};");
26           writeln("cont_substate" + i + ": boolean;");
27           writeln("end_substate" + i + ": boolean;");
28         }
29       }
30     }
```

Figure B.22: The gen_var_decl method

B.4 Example 2

To further illustrate the process reported in the thesis and the tools that have been developed to support it, this section provides another example starting from a diagram that depicts a range of the features of UML statecharts, so that the way each of these features is handled at the different stages can be observed.

B.4.1 Statechart Diagrams

As discussed in Chapter 3, the features of UML statecharts that are addressed in this thesis are:

- event
- state

```
31      else {
32        write("substate: ");
33        write("{");
34        index = 1;
35        substate_var_decl.add("nil");
36        it = substate_var_decl.iterator();
37        while (it.hasNext()) {
38          if (index == 1) {
39            write(it.next());
40          }
41          else {
42            write("," + it.next());
43          }
44          index++;
45        }
46        writeln("};");
47      }
48      if (!concur_comp_state_flag) {
49        writeln(ack_decl);
50      }
51      retrieve_guard_cond_decl();
52      writeln(superstate_enable_decl);
53    }
54  }
```

Figure B.23: The gen_var_decl method (continued)

```
1  private static void gen_assign_stmts() {
2    writeln("ASSIGN");
3    retrieve_case_stmt();
4  }
```

Figure B.24: The gen_assign_stmts method

- guard-condition
- action
- non-concurrent composite state
- concurrent composite state

Thus the diagram (taken from a screenshot of the Poseidon for UML system) in Figure B.27 has been prepared to capture features event, state, guard-condition, action and non-concurrent composite state. Poseidon for UML exports a statechart diagram in an XMI format and the equivalent of the diagram in the screenshot (with elision to avoid unnecessary detail) appears in Figures B.28, B.29 and B.30. The XMI representations of the states S_1 and S_2 as well as the non-composite state S_3 and its

```
1  public static void retrieve_case_stmt() {
2    int k;
3    String prev_var = "";
4    String tmp_substate_stmts = "";
5    Iterator it1 = var_list.iterator();
6    while (it1.hasNext()) {
7      String cur_var = (String) it1.next();
8      ListIterator it2 = case_cond_varname.listIterator();
9      int index = 0;
10     while (it2.hasNext()) {
11       if (cur_var.equals((String)it2.next())) {
12         ListIterator it3 = case_cond_type.listIterator(index);
13         ListIterator it4 = case_cond_stmt.listIterator(index);
14         String out_type = (String)it3.next();
15         String out_stmt = (String)it4.next();
16         if (!cur_var.equals(prev_var)) {
17           if (cur_var.equals("state"))
18             gen_init_state();
19           if (cur_var.equals("substate"))
20             gen_init_substate();
21           if (cur_var.endsWith("_sent"))
22             gen_init_sent(cur_var);
23           if (cur_var.startsWith("substate") & cur_var.length() > 8)
24             gen_init_substate_of_concur(cur_var);
25           if (cur_var.startsWith("cont_substate"))
26             gen_init_cont_substate(cur_var);
27           if (cur_var.startsWith("end_substate"))
28             gen_init_end_substate(cur_var);
29           writeln(out_type + "(" + cur_var +") " + ":= ");
30           writeln("case");
31         }
32         writeln(out_stmt);
33         prev_var = cur_var;
34       }
35       index++;
36     }
```

Figure B.25: The retrieve_case_stmt method

```
37      if (cur_var.equals("step")) {
38        if (substate_var_decl.isEmpty())
39          writeln("state=next(state) & !step: 1" + ";");
40        else {
41          Iterator it5 = superstate_list.iterator();
42          while (it5.hasNext()) {
43            writeln("state=" + it5.next() + " & ack=neg & !step " +
44                    "& superstate_enable: 1;");
45          }
46          if (no_of_regions > 1) {
47            for (int m=1; m<=no_of_regions; m++) {
48              tmp_substate_stmts = tmp_substate_stmts + " & substate" +
49                                   m + " = nil";
50            }
51            writeln("state=next(state) & !step" + tmp_substate_stmts +
52                    ": 1" + ";");
53          }
54          else {
55            writeln("state=next(state) & !step & substate = nil: 1" + ";");
56            writeln("state=next(state) & !step & superstate_enable: 1" + ";");
57          }
58        }
59        writeln("event_buff != empty & step: 0;");
60      }
61      if (cur_var.equals("ack")) {
62        if (!case_cond_varname.contains("ack")) {
63          writeln("next" + "(" + cur_var +") " + ":= ");
64          writeln("case");
65        }
66        writeln("substate=next(substate) & !step & !superstate_enable " +
67                "& !substate = nil: neg;");
68        writeln("1: " + "undefine" + ";");
69      }
70      else {
71        if (cur_var.startsWith("ack_substate")) {
72          k = Integer.parseInt(cur_var.substring(12));
73          writeln("substate" + k + " = next(substate" + k + ") & !step & " +
74            "!superstate_enable & !substate" + k + " = nil: neg;");
75          writeln("1: undefine;");
76        }
77        else {
78          writeln("1: " + cur_var + ";");
79        }
80      }
81      writeln("esac;");
82    }
83    if (!substate_var_decl.isEmpty())
84      gen_superstate_enable_case_stmts();
85    gen_event_buff_stmts();
86  }
```

Figure B.26: The retrieve_case_stmt method (continued)

substate V_1 are on lines 4–16, 17–32, 33–58 and 44–56, respectively. Similarly, lines 61–109, 110–128, 131–140 and 141–150 are the equivalent representations of the two transitions and the two events in XMI format.

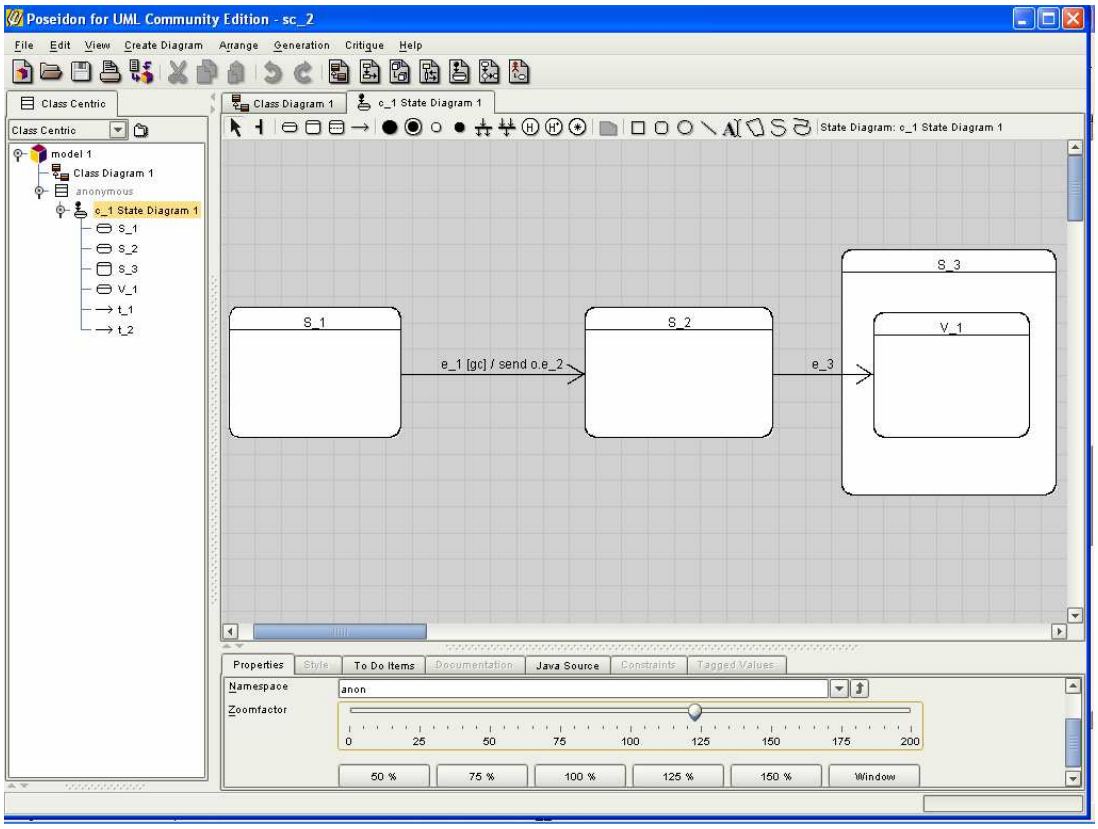

Figure B.27: The screenshot of example 2

B.4.2 The π-Calculus Representation

The first step is to translate the XMI representation of the UML statechart into the π-calculus, using the program SC2PiCal. Applying the translation rules that were discussed in detail in Chapter 3, the specification in Figure B.31 is obtained. This π-calculus notation is written to satisfy the input grammar for the MWB (Figure B.31); a more conventional presentation is given in Figures B.32 and B.33 . Particular points to observe about the translation are:

lines 122–127 (Figure B.4): A non-concurrent composite state is distinguished from a concurrent composite state by the *isConcurrent* attribute as shown on line 34

```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Fri Nov 25 00:08:41 CST 2005'>
...
<UML:SimpleState xmi.id = 'a13' name = 'S_1' isSpecification = 'false'>
  <UML:ModelElement.taggedValue>
    <UML:TaggedValue xmi.id = 'a14' isSpecification = 'false'
                     dataValue = '-64--88-1-100-114382d:107c2f610ba:-7ffc'>
      <UML:TaggedValue.type>
        <UML:TagDefinition xmi.idref = 'a6'/>
      </UML:TaggedValue.type>
    </UML:TaggedValue>
  </UML:ModelElement.taggedValue>
    <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref = 'a15'/>
  </UML:StateVertex.outgoing>
</UML:SimpleState>
<UML:SimpleState xmi.id = 'a16' name = 'S_2' isSpecification = 'false'>
  <UML:ModelElement.taggedValue>
    <UML:TaggedValue xmi.id = 'a17' isSpecification = 'false'
                     dataValue = '-64--88-1-100-114382d:107c2f610ba:-7ffb'>
      <UML:TaggedValue.type>
        <UML:TagDefinition xmi.idref = 'a6'/>
      </UML:TaggedValue.type>
    </UML:TaggedValue>
  </UML:ModelElement.taggedValue>
  <UML:StateVertex.outgoing>
    <UML:Transition xmi.idref = 'a18'/>
  </UML:StateVertex.outgoing>
  <UML:StateVertex.incoming>
    <UML:Transition xmi.idref = 'a15'/>
  </UML:StateVertex.incoming>
</UML:SimpleState>
<UML:CompositeState xmi.id = 'a19' name = 'S_3' isSpecification = 'false'
  isConcurrent = 'false'>
  <UML:ModelElement.taggedValue>
    <UML:TaggedValue xmi.id = 'a20' isSpecification = 'false'
                     dataValue = '-64--88-1-100-114382d:107c2f610ba:-7ffa'>
      <UML:TaggedValue.type>
        <UML:TagDefinition xmi.idref = 'a6'/>
      </UML:TaggedValue.type>
    </UML:TaggedValue>
  </UML:ModelElement.taggedValue>
  <UML:CompositeState.subvertex>
    <UML:SimpleState xmi.id = 'a21' name = 'V_1' isSpecification = 'false'>
      <UML:ModelElement.taggedValue>
        <UML:TaggedValue xmi.id = 'a22' isSpecification = 'false'
                         dataValue = '-64--88-1-100-114382d:107c2f610ba:-7ff9'>
          <UML:TaggedValue.type>
            <UML:TagDefinition xmi.idref = 'a6'/>
          </UML:TaggedValue.type>
        </UML:TaggedValue>
      </UML:ModelElement.taggedValue>
      <UML:StateVertex.incoming>
        <UML:Transition xmi.idref = 'a18'/>
      </UML:StateVertex.incoming>
    </UML:SimpleState>
  </UML:CompositeState.subvertex>
</UML:CompositeState>
...
<UML:StateMachine.transitions>
  <UML:Transition xmi.id = 'a15' name = 't_1' isSpecification = 'false'>
    <UML:ModelElement.taggedValue>
      <UML:TaggedValue xmi.id = 'a23' isSpecification = 'false'
                       dataValue = '-64--88-1-100-114382d:107c2f610ba:-7ff6'>
        <UML:TaggedValue.type>
          <UML:TagDefinition xmi.idref = 'a6'/>
        </UML:TaggedValue.type>
      </UML:TaggedValue>
    </UML:ModelElement.taggedValue>
    <UML:Transition.guard>
```

Figure B.28: The XMI rendering of example 2

```
71          <UML:Guard xmi.id = 'a24' name = 'g1' isSpecification = 'false'>
72            <UML:Guard.expression>
73              <UML:BooleanExpression xmi.id = 'a25' language = 'java' body = 'gc'/>
74            </UML:Guard.expression>
75            <UML:ModelElement.taggedValue>
76              <UML:TaggedValue xmi.id = 'a26' isSpecification = 'false'
77                               dataValue = '-64--88-1-100-114382d:107c2f610ba:-7ff3'>
78                <UML:TaggedValue.type>
79                  <UML:TagDefinition xmi.idref = 'a6'/>
80                </UML:TaggedValue.type>
81              </UML:TaggedValue>
82            </UML:ModelElement.taggedValue>
83          </UML:Guard>
84        </UML:Transition.guard>
85        <UML:Transition.effect>
86          <UML:CallAction xmi.id = 'a27' name = 'a_1' isSpecification = 'false' isAsynchronous = 'false'>
87            <UML:Action.script>
88              <UML:ActionExpression xmi.id = 'a28' language = 'java' body = 'send o.e_2'/>
89            </UML:Action.script>
90            <UML:ModelElement.taggedValue>
91              <UML:TaggedValue xmi.id = 'a29' isSpecification = 'false'
92                               dataValue = '-64--88-1-100-114382d:107c2f610ba:-7ff1'>
93                <UML:TaggedValue.type>
94                  <UML:TagDefinition xmi.idref = 'a6'/>
95                </UML:TaggedValue.type>
96              </UML:TaggedValue>
97            </UML:ModelElement.taggedValue>
98          </UML:CallAction>
99        </UML:Transition.effect>
100       <UML:Transition.trigger>
101         <UML:CallEvent xmi.idref = 'a30'/>
102       </UML:Transition.trigger>
103       <UML:Transition.source>
104         <UML:SimpleState xmi.idref = 'a13'/>
105       </UML:Transition.source>
106       <UML:Transition.target>
107         <UML:SimpleState xmi.idref = 'a16'/>
108       </UML:Transition.target>
109     </UML:Transition>
110     <UML:Transition xmi.id = 'a18' name = 't_2' isSpecification = 'false'>
111       <UML:ModelElement.taggedValue>
112         <UML:TaggedValue xmi.id = 'a31' isSpecification = 'false'
113                          dataValue = '-64--88-1-100-114382d:107c2f610ba:-7ff5'>
114           <UML:TaggedValue.type>
115             <UML:TagDefinition xmi.idref = 'a6'/>
116           </UML:TaggedValue.type>
117         </UML:TaggedValue>
118       </UML:ModelElement.taggedValue>
119       <UML:Transition.trigger>
120         <UML:CallEvent xmi.idref = 'a32'/>
121       </UML:Transition.trigger>
122       <UML:Transition.source>
123         <UML:SimpleState xmi.idref = 'a16'/>
124       </UML:Transition.source>
125       <UML:Transition.target>
126         <UML:SimpleState xmi.idref = 'a21'/>
127       </UML:Transition.target>
128     </UML:Transition>
129   </UML:StateMachine.transitions>
130   ...
131   <UML:CallEvent xmi.id = 'a30' name = 'e_1' isSpecification = 'false'>
132     <UML:ModelElement.taggedValue>
133       <UML:TaggedValue xmi.id = 'a33' isSpecification = 'false'
134                        dataValue = '-64--88-1-100-114382d:107c2f610ba:-7ff4'>
135         <UML:TaggedValue.type>
136           <UML:TagDefinition xmi.idref = 'a6'/>
137         </UML:TaggedValue.type>
138       </UML:TaggedValue>
139     </UML:ModelElement.taggedValue>
140   </UML:CallEvent>
```

Figure B.29: The XMI rendering of example 2 (continued)

```
141 <UML:CallEvent xmi.id = 'a32' name = 'e_3' isSpecification = 'false'>
142   <UML:ModelElement.taggedValue>
143     <UML:TaggedValue xmi.id = 'a34' isSpecification = 'false'
144                      dataValue = '-64--88-1-100-114382d:107c2f610ba:-7ff0'>
145       <UML:TaggedValue.type>
146         <UML:TagDefinition xmi.idref = 'a6'/>
147       </UML:TaggedValue.type>
148     </UML:TaggedValue>
149   </UML:ModelElement.taggedValue>
150 </UML:CallEvent>
151 ...
152 </XMI>
```

Figure B.30: The XMI rendering of example 2 (continued)

of Figure B.28. Line 125 of Figure B.4 sets the Boolean variable *concurrent* to the value *true* if the composite state is a concurrent composite state.

lines 3–8 (Figure B.15): If the target state is a substate which is enclosed by a non-concurrent composite state, lines 5–8 of Figure B.15 output the channel declaration on line 15 and the process identifiers on lines 16 and 17 of Figure B.31.

```
1  agent S_1(step,event_ch0,e_1,e_3,gc,ins_o,e_2,pos,neg)= \
2  event_ch0(x1). \
3  ([x1=e_1] \
4  (^true,false)'gc<true,false>. \
5  (true. \
6  'ins_o<e_2>. \
7  'step.S_2(step,event_ch0,e_1,e_3,gc,ins_o,e_2,pos,neg)+ \
8   false. \
9  'step.S_1(step,event_ch0,e_1,e_3,gc,ins_o,e_2,pos,neg))+ \
10 [x1=e_3]'step.S_1(step,event_ch0,e_1,e_3,gc,ins_o,e_2,pos,neg))
11
12 agent S_2(step,event_ch0,e_1,e_3,gc,ins_o,e_2,pos,neg)= \
13 event_ch0(x1). \
14 ([x1=e_3] \
15 'step.(^event_ch1) \
16 (S_3(step,event_ch0,event_ch1,e_1,e_3,gc,ins_o,e_2,pos,neg) | \
17 V_1(event_ch1,e_1,e_3,gc,ins_o,e_2,pos,neg))+ \
18 [x1=e_1]'step.S_2(step,event_ch0,e_1,e_3,gc,ins_o,e_2,pos,neg))
```

Figure B.31: The machine-readable form of the π-calculus representation of the XMI rendering in Figures B.28–B.30

B.4.3 The NuSMV Representation

In the second part of the thesis, the objective was to establish properties of the UML statechart using model-checking by means of the NuSMV model-checker. This rep-

$$
\begin{aligned}
&S\_1(step, event_{ch0}, e\_1, e\_3, gc, ins_o, e\_2, pos, neg) = \\
&\quad event_{ch0}(x1). \\
&\quad ([x1 = e\_1] \\
&\quad (\boldsymbol{\nu} true\ false)\overline{gc}\langle true\ false\rangle. \\
&\quad (true. \\
&\quad \overline{ins_o}\langle e\_2\rangle. \\
&\quad \overline{step}.S\_2(step, event_{ch0}, e\_1, e\_3, gc, ins_o, e\_2, pos, neg) + \\
&\quad false. \\
&\quad \overline{step}.S\_1(step, event_{ch0}, e\_1, e\_3, gc, ins_o, e\_2, pos, neg)) + \\
&\quad [x1 = e\_3]\overline{step}.S\_1(step, event_{ch0}, e\_1, e\_3, gc, ins_o, e\_2, pos, neg))
\end{aligned}
$$

Figure B.32: The pretty-printed form of the π-calculus in Figure B.31

$$
\begin{aligned}
&S\_2(step, event_{ch0}, e\_1, e\_3, gc, ins_o, e\_2, pos, neg) = \\
&\quad event_{ch0}(x1). \\
&\quad ([x1 = e\_3] \\
&\quad \overline{step}.(\boldsymbol{\nu} event_{ch1}) \\
&\quad (S\_3(step, event_{ch0}, event_{ch1}, e\_1, e\_3, gc, ins_o, e\_2, pos, neg)| \\
&\quad V\_1(event_{ch1}, e\_1, e\_3, gc, ins_o, e\_2, pos, neg)) + \\
&\quad [x1 = e\_1]\overline{step}.S\_2(step, event_{ch0}, e\_1, e\_3, gc, ins_o, e\_2, pos, neg))
\end{aligned}
$$

Figure B.33: The pretty-printed form of the π-calculus in Figure B.31 (continued)

resentation (Figures B.34 and B.35) is generated according to the rules presented in Chapter 5. The main features of note in the translation are:

lines 32–46, 49 and 52 (Figure B.23): If a statechart diagram contains a non-concurrent composite state, NuSMV variables *substate*, *ack* and *superstate_enable* are generated as shown on lines 4, 5 and 7 of Figure B.34.

lines 61–69 (Figure B.26): The code fragment yields the next statements for the

NuSMV variable *ack* on lines 14–18 of Figure B.34.

line 84 (Figure B.26): The invocation of the method *gen_superstate_enable_case_stmts* emits lines 48–53 of Figure B.35.

```
1  MODULE sc_2(ins_o,o_q_buff,step,event_buff)
2  VAR
3  state: {S_3,S_1,S_2};
4  substate: {nil,V_1};
5  ack: {pos, neg, undefine};
6  gc: boolean;
7  superstate_enable: boolean;
8  ASSIGN
9  next(ins_o) :=
10 case
11 state=S_1 & event_buff=e_1 & gc & !ins_o & !step:1;
12 1: ins_o;
13 esac;
14 next(ack) :=
15 case
16 substate=next(substate) & !step & !superstate_enable & !substate = nil: neg;
17 1: undefine;
18 esac;
```

Figure B.34: The NuSMV representation of the π-calculus in Figure B.31

B.5 Example 3

Likewise, the UML statechart in Figure B.36 is represented in XMI format as shown in Figures B.37–B.40. The value of the *isConcurrent* attribute is *true* (line 5) as state *S_2* is a concurrent composite state. The XMI representations of the composite state and basic states are on lines 4–108. Similarly, the XMI representations of the transitions and events are on lines 110–202 and 204–243, respectively. The corresponding π-calculus representation and NuSMV code have been provided and discussed in detail in Section 6.8. A few additional remarks are given below:

lines 128–146 (Figure B.5): For each attribute (line 137) of the grandparent node of the owner element (line 135), the attribute name and attribute value are tested. If the attribute name and attribute value equal to *isConcurrent* and *true* (lines 139 and 140), respectively, this means that the current node name is the name of a substate of a concurrent composite state. The substate name of the concurrent

```
19 init(substate) := nil;
20 next(substate) :=
21 case
22 state=S_2 & event_buff=e_3 & !step:V_1;
23 1: substate;
24 esac;
25 init(state) := S_1;
26 next(state) :=
27 case
28 state=S_1 & event_buff=e_1 & gc & !ins_o & !step:S_2;
29 state=S_2 & event_buff=e_3 & !step:S_3;
30 1: state;
31 esac;
32 next(o_q_buff) :=
33 case
34 state=S_1 & event_buff=e_1 & gc & !ins_o & !step:e_2;
35 1: o_q_buff;
36 esac;
37 next(step) :=
38 case
39 state=S_1 & event_buff=e_1 & gc & !ins_o & !step:1;
40 state=S_1 & event_buff=e_1 & gc & ins_o & !step:step;
41 state=S_2 & event_buff=e_3 & !step:1;
42 state=S_3 & ack=neg & !step & superstate_enable: 1;
43 state=next(state) & !step & substate = nil: 1;
44 state=next(state) & !step & superstate_enable: 1;
45 event_buff != empty & step: 0;
46 1: step;
47 esac;
48 init(superstate_enable) := 0;
49 next(superstate_enable) :=
50 case
51 (next(ack)=pos | next(ack)=neg) & !step: 1;
52 1: 0;
53 esac;
54 next(event_buff):=
55 case
56 event_buff != empty & next(step): empty;
57 1 : event_buff;
58 esac;
59 FAIRNESS running
```

Figure B.35: The NuSMV representation of the π-calculus in Figure B.31 (continued)

composite state is then kept in a set *concur_substate* (line 141) which differentiates it from other states.

lines 6–8 (Figure B.13): A snippet which yields an input action when the source and target states are substates of a concurrent composite state. States V_1 and V_2 as well as W_1 and W_2 are examples of this.

lines 71–73 (Figure B.16): The transition which connects the state S_1 and the fork pseudostate is handled by this block of code.

lines 77–85 (Figure B.16): A code fragment that deals with the join pseudostate in Figure B.36.

lines 10–15 (Figure B.17): These statements are executed and they produce matching constructs for other events when both the source state (transition[index][0]) and target state (transition[index][4]) are substates of a concurrent composite state. The map *concur_ch_table* (lines 12 and 13) associates a substate with its corresponding channel for receiving an event. An example of which is the states V_1 and V_2 of Figure B.36.

lines 18–22 (Figure B.17): An execution of the code fragment outputs matching constructs for other events if the source state (transition[index][0]) is a substate of a concurrent composite state and the target state (transition[index][4]) is a join. A typical example is the transition between the state V_2 and the join in Figure B.36.

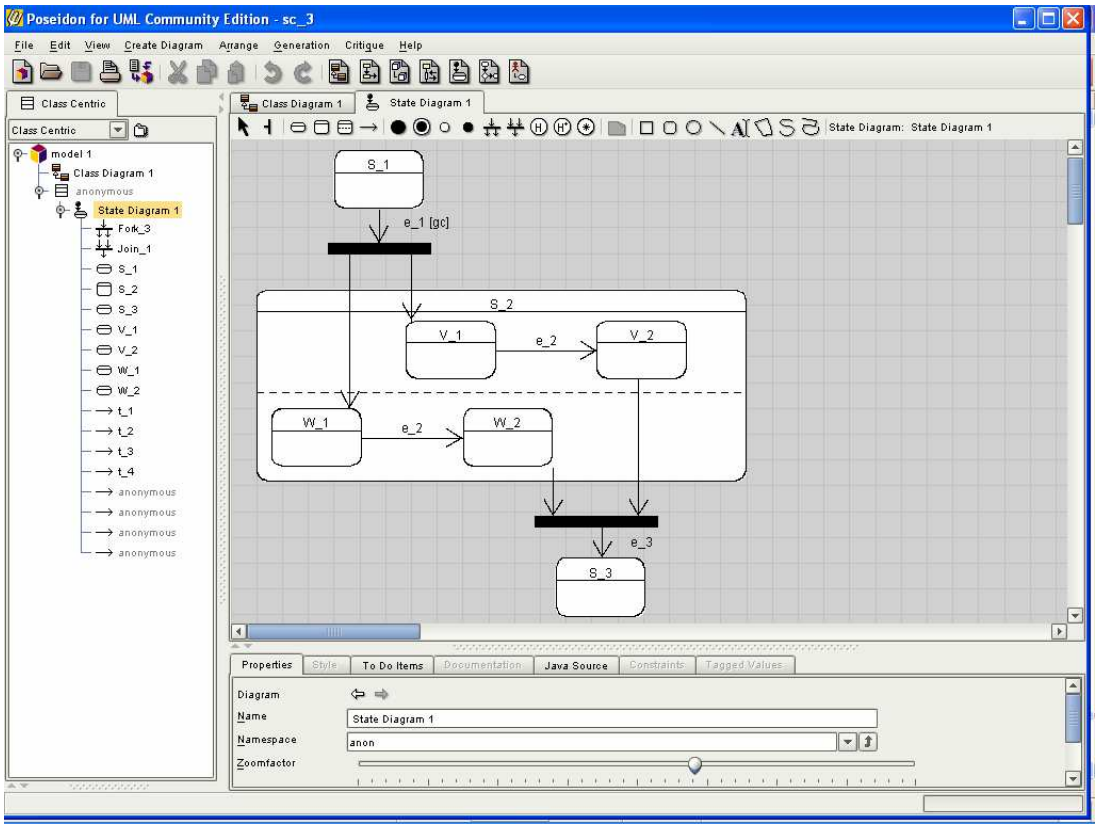

Figure B.36: The screenshot of example 3

```
<?xml version = '1.0' encoding = 'UTF-8' ?>
<XMI xmi.version = '1.2' xmlns:UML = 'org.omg.xmi.namespace.UML' timestamp = 'Mon Nov 28 13:42:36 CST 2005'>
...
<UML:CompositeState xmi.id = 'a11' name = 'S_2' isSpecification = 'false'
  isConcurrent = 'true'>
  <UML:ModelElement.taggedValue>
    <UML:TaggedValue xmi.id = 'a12' isSpecification = 'false'
                     dataValue = '-64--88-1-100-779dce:107d54ed365:-7fea'>
      <UML:TaggedValue.type>
        <UML:TagDefinition xmi.idref = 'a13'/>
      </UML:TaggedValue.type>
    </UML:TaggedValue>
  </UML:ModelElement.taggedValue>
  <UML:CompositeState.subvertex>
    <UML:SimpleState xmi.id = 'a14' name = 'W_1' isSpecification = 'false'>
      <UML:ModelElement.taggedValue>
        <UML:TaggedValue xmi.id = 'a15' isSpecification = 'false'
                         dataValue = '-64--88-1-100-779dce:107d54ed365:-7fdb'>
          <UML:TaggedValue.type>
            <UML:TagDefinition xmi.idref = 'a13'/>
          </UML:TaggedValue.type>
        </UML:TaggedValue>
      </UML:ModelElement.taggedValue>
```

Figure B.37: The XMI representation of example 3

```
24        <UML:StateVertex.outgoing>
25          <UML:Transition xmi.idref = 'a16'/>
26        </UML:StateVertex.outgoing>
27        <UML:StateVertex.incoming>
28          <UML:Transition xmi.idref = 'a17'/>
29        </UML:StateVertex.incoming>
30      </UML:SimpleState>
31      <UML:SimpleState xmi.id = 'a18' name = 'V_1' isSpecification = 'false'>
32        <UML:ModelElement.taggedValue>
33          <UML:TaggedValue xmi.id = 'a19' isSpecification = 'false'
34                           dataValue = '-64--88-1-100-779dce:107d54ed365:-7fda'>
35            <UML:TaggedValue.type>
36              <UML:TagDefinition xmi.idref = 'a13'/>
37            </UML:TaggedValue.type>
38          </UML:TaggedValue>
39        </UML:ModelElement.taggedValue>
40        <UML:StateVertex.outgoing>
41          <UML:Transition xmi.idref = 'a20'/>
42        </UML:StateVertex.outgoing>
43        <UML:StateVertex.incoming>
44          <UML:Transition xmi.idref = 'a21'/>
45        </UML:StateVertex.incoming>
46      </UML:SimpleState>
47      <UML:SimpleState xmi.id = 'a22' name = 'V_2' isSpecification = 'false'>
48        <UML:ModelElement.taggedValue>
49          <UML:TaggedValue xmi.id = 'a23' isSpecification = 'false'
50                           dataValue = '-64--88-1-100-779dce:107d54ed365:-7fd9'>
51            <UML:TaggedValue.type>
52              <UML:TagDefinition xmi.idref = 'a13'/>
53            </UML:TaggedValue.type>
54          </UML:TaggedValue>
55        </UML:ModelElement.taggedValue>
56        <UML:StateVertex.outgoing>
57          <UML:Transition xmi.idref = 'a24'/>
58        </UML:StateVertex.outgoing>
59        <UML:StateVertex.incoming>
60          <UML:Transition xmi.idref = 'a20'/>
61        </UML:StateVertex.incoming>
62      </UML:SimpleState>
63      <UML:SimpleState xmi.id = 'a25' name = 'W_2' isSpecification = 'false'>
64        <UML:ModelElement.taggedValue>
65          <UML:TaggedValue xmi.id = 'a26' isSpecification = 'false'
66                           dataValue = '-64--88-1-100-779dce:107d54ed365:-7fd8'>
67            <UML:TaggedValue.type>
68              <UML:TagDefinition xmi.idref = 'a13'/>
69            </UML:TaggedValue.type>
70          </UML:TaggedValue>
71        </UML:ModelElement.taggedValue>
72        <UML:StateVertex.outgoing>
73          <UML:Transition xmi.idref = 'a27'/>
74        </UML:StateVertex.outgoing>
75        <UML:StateVertex.incoming>
76          <UML:Transition xmi.idref = 'a16'/>
77        </UML:StateVertex.incoming>
78      </UML:SimpleState>
79    </UML:CompositeState.subvertex>
80  </UML:CompositeState>
81  ...
82  <UML:SimpleState xmi.id = 'a31' name = 'S_1' isSpecification = 'false'>
83    <UML:ModelElement.taggedValue>
84      <UML:TaggedValue xmi.id = 'a32' isSpecification = 'false'
85                       dataValue = '-64--88-1-100-779dce:107d54ed365:-7fe4'>
86        <UML:TaggedValue.type>
87          <UML:TagDefinition xmi.idref = 'a13'/>
88        </UML:TaggedValue.type>
89      </UML:TaggedValue>
90    </UML:ModelElement.taggedValue>
91    <UML:StateVertex.outgoing>
92      <UML:Transition xmi.idref = 'a30'/>
93    </UML:StateVertex.outgoing>
94  </UML:SimpleState>
95  ...
96  <UML:SimpleState xmi.id = 'a36' name = 'S_3' isSpecification = 'false'>
97    <UML:ModelElement.taggedValue>
98      <UML:TaggedValue xmi.id = 'a37' isSpecification = 'false'
99                       dataValue = '-64--88-1-100-779dce:107d54ed365:-7fd5'>
100       <UML:TaggedValue.type>
101         <UML:TagDefinition xmi.idref = 'a13'/>
102       </UML:TaggedValue.type>
103     </UML:TaggedValue>
104   </UML:ModelElement.taggedValue>
105   <UML:StateVertex.incoming>
106     <UML:Transition xmi.idref = 'a35'/>
107   </UML:StateVertex.incoming>
108 </UML:SimpleState>
109 ...
```

Figure B.38: The XMI representation of example 3 (continued)

```
<UML:Transition xmi.id = 'a30' name = 't_1' isSpecification = 'false'>
  <UML:ModelElement.taggedValue>
    <UML:TaggedValue xmi.id = 'a38' isSpecification = 'false'
                     dataValue = '-64--88-1-100-779dce:107d54ed365:-7fe2'>
      <UML:TaggedValue.type>
        <UML:TagDefinition xmi.idref = 'a13'/>
      </UML:TaggedValue.type>
    </UML:TaggedValue>
  </UML:ModelElement.taggedValue>
  <UML:Transition.guard>
    <UML:Guard xmi.id = 'a39' name = 'anon' isSpecification = 'false'>
      <UML:Guard.expression>
        <UML:BooleanExpression xmi.id = 'a40' language = 'java' body = 'gc'/>
      </UML:Guard.expression>
      <UML:ModelElement.taggedValue>
        <UML:TaggedValue xmi.id = 'a41' isSpecification = 'false'
                         dataValue = '-64--88-1-100-779dce:107d54ed365:-7fc9'>
          <UML:TaggedValue.type>
            <UML:TagDefinition xmi.idref = 'a13'/>
          </UML:TaggedValue.type>
        </UML:TaggedValue>
      </UML:ModelElement.taggedValue>
    </UML:Guard>
  </UML:Transition.guard>
  <UML:Transition.trigger>
    <UML:CallEvent xmi.idref = 'a42'/>
  </UML:Transition.trigger>
  <UML:Transition.source>
    <UML:SimpleState xmi.idref = 'a31'/>
  </UML:Transition.source>
  <UML:Transition.target>
    <UML:Pseudostate xmi.idref = 'a28'/>
  </UML:Transition.target>
</UML:Transition>
...
<UML:Transition xmi.id = 'a35' name = 't_4' isSpecification = 'false'>
  <UML:ModelElement.taggedValue>
    <UML:TaggedValue xmi.id = 'a45' isSpecification = 'false'
                     dataValue = '-64--88-1-100-779dce:107d54ed365:-7fd4'>
      <UML:TaggedValue.type>
        <UML:TagDefinition xmi.idref = 'a13'/>
      </UML:TaggedValue.type>
    </UML:TaggedValue>
  </UML:ModelElement.taggedValue>
  <UML:Transition.trigger>
    <UML:CallEvent xmi.idref = 'a46'/>
  </UML:Transition.trigger>
  <UML:Transition.source>
    <UML:Pseudostate xmi.idref = 'a33'/>
  </UML:Transition.source>
  <UML:Transition.target>
    <UML:SimpleState xmi.idref = 'a36'/>
  </UML:Transition.target>
</UML:Transition>
...
<UML:Transition xmi.id = 'a20' name = 't_2' isSpecification = 'false'>
  <UML:ModelElement.taggedValue>
    <UML:TaggedValue xmi.id = 'a49' isSpecification = 'false'
                     dataValue = '-64--88-1-100-779dce:107d54ed365:-7fcf'>
      <UML:TaggedValue.type>
        <UML:TagDefinition xmi.idref = 'a13'/>
      </UML:TaggedValue.type>
    </UML:TaggedValue>
  </UML:ModelElement.taggedValue>
```

Figure B.39: The XMI representation of example 3 (continued)

```
174     <UML:Transition.trigger>
175       <UML:CallEvent xmi.idref = 'a50'/>
176     </UML:Transition.trigger>
177     <UML:Transition.source>
178       <UML:SimpleState xmi.idref = 'a18'/>
179     </UML:Transition.source>
180     <UML:Transition.target>
181       <UML:SimpleState xmi.idref = 'a22'/>
182     </UML:Transition.target>
183   </UML:Transition>
184   <UML:Transition xmi.id = 'a16' name = 't_3' isSpecification = 'false'>
185     <UML:ModelElement.taggedValue>
186       <UML:TaggedValue xmi.id = 'a51' isSpecification = 'false'
187                        dataValue = '-64--88-1-100-779dce:107d54ed365:-7fce'>
188         <UML:TaggedValue.type>
189           <UML:TagDefinition xmi.idref = 'a13'/>
190         </UML:TaggedValue.type>
191       </UML:TaggedValue>
192     </UML:ModelElement.taggedValue>
193     <UML:Transition.trigger>
194       <UML:CallEvent xmi.idref = 'a52'/>
195     </UML:Transition.trigger>
196     <UML:Transition.source>
197       <UML:SimpleState xmi.idref = 'a14'/>
198     </UML:Transition.source>
199     <UML:Transition.target>
200       <UML:SimpleState xmi.idref = 'a25'/>
201     </UML:Transition.target>
202   </UML:Transition>
203   ...
204   <UML:CallEvent xmi.id = 'a42' name = 'e_1' isSpecification = 'false'>
205     <UML:ModelElement.taggedValue>
206       <UML:TaggedValue xmi.id = 'a53' isSpecification = 'false'
207                        dataValue = '-64--88-1-100-779dce:107d54ed365:-7fcd'>
208         <UML:TaggedValue.type>
209           <UML:TagDefinition xmi.idref = 'a13'/>
210         </UML:TaggedValue.type>
211       </UML:TaggedValue>
212     </UML:ModelElement.taggedValue>
213   </UML:CallEvent>
214   <UML:CallEvent xmi.id = 'a50' name = 'e_2' isSpecification = 'false'>
215     <UML:ModelElement.taggedValue>
216       <UML:TaggedValue xmi.id = 'a54' isSpecification = 'false'
217                        dataValue = '-64--88-1-100-779dce:107d54ed365:-7fcc'>
218         <UML:TaggedValue.type>
219           <UML:TagDefinition xmi.idref = 'a13'/>
220         </UML:TaggedValue.type>
221       </UML:TaggedValue>
222     </UML:ModelElement.taggedValue>
223   </UML:CallEvent>
224   <UML:CallEvent xmi.id = 'a52' name = 'e_2' isSpecification = 'false'>
225     <UML:ModelElement.taggedValue>
226       <UML:TaggedValue xmi.id = 'a55' isSpecification = 'false'
227                        dataValue = '-64--88-1-100-779dce:107d54ed365:-7fcb'>
228         <UML:TaggedValue.type>
229           <UML:TagDefinition xmi.idref = 'a13'/>
230         </UML:TaggedValue.type>
231       </UML:TaggedValue>
232     </UML:ModelElement.taggedValue>
233   </UML:CallEvent>
234   <UML:CallEvent xmi.id = 'a46' name = 'e_3' isSpecification = 'false'>
235     <UML:ModelElement.taggedValue>
236       <UML:TaggedValue xmi.id = 'a56' isSpecification = 'false'
237                        dataValue = '-64--88-1-100-779dce:107d54ed365:-7fca'>
238         <UML:TaggedValue.type>
239           <UML:TagDefinition xmi.idref = 'a13'/>
240         </UML:TaggedValue.type>
241       </UML:TaggedValue>
242     </UML:ModelElement.taggedValue>
243   </UML:CallEvent>
244   ...
245   </XMI>
```

Figure B.40: The XMI representation of example 3 (continued)

Bibliography

[1] E. Armstrong, S. Bodoff, D. Carson, M. Fisher, S. Fordin, D. Green, K. Haase, and E. Jendrock. *The Java Web Services Tutorial*, February 2003. `http://java.sun.com/webservices/docs/1.1/tutorial/doc/index.html`; accessed January 20, 2005.

[2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, third edition, 2000.

[3] N. Asokan, P.A. Janson, M. Steiner, and M. Waider. The state of the art in electronic payment systems. *IEEE Computer*, 30(9):28–35, 1997.

[4] B. Bauer, J.P. Müller, and J. Odell. Agent UML: a formalism for specifying multiagent software systems. In *Proceedings of the First International Workshop AOSE 2000*, LNCS 1957, pages 91–104, 2001.

[5] M. Bellare, J.A. Garay, R. Hauser, A. Herzberg, H. Krawczyk, and M. Steiner. Design, implementation, and deployment of the iKP secure electronic payment system. *IEEE Journal on Selected Areas in Communications*, 18(4):611–627, 2000.

[6] M. Ben-Ari. *Mathematical Logic for Computer Science*. Prentice Hall, 1993.

[7] J. Bennett. *Introduction to Compiler Techniques: A first course using ANSI C, LEX and YACC*. McGraw-Hill, 1990.

[8] M. Berger and K. Honda. The two-phase commit protocol in an extended π-calculus. *Electronic Notes in Theoretical Computer Science*, 39(1):21–46, 2000.

[9] J.A. Bergstra, A. Ponse, and S.A. Smolka. *Handbook of Process Algebra*. Elsevier Science, 2001.

[10] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[11] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *36th ACM/IEEE Conference on Design Automation (DAC '99)*, pages 317–320, 1999.

[12] A. Biere, A. Cimatti, E.M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1579, pages 193–207, 1999.

[13] M. Boger, T. Sturm, and E. Schildhauer. *Poseidon for UML Users Guide*. Gentleware AG, 2002. `http://www.gentleware.com`; accessed January 20, 2005.

[14] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.

[15] A. Bouali, S. Gnesi, and S. Larosa. The integration project for the JACK environment. *Bulletin of the EATCS*, 54:207–223, 1994. `http://matrix.iei.pi.cnr.it/projects/JACK`; accessed January 20, 2005.

[16] J.M. Bradshaw, editor. *Software Agents*. AAAI Press/The MIT Press, 1997.

[17] P. Braun, H. Lötzbeyer, B. Schätz, and O. Slotosch. Consistent integration of formal methods. In *TACAS 2000*, LNCS 1785, pages 48–62, 2000.

[18] T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler. *Extensible Markup Language (XML) 1.0*. W3C, second edition, October 2000. `http://www.w3.org/XML`; accessed January 20, 2005.

[19] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[20] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[21] M. Buchholtz, H.R. Nielson, and F. Nielson. *Deliverable D19: Static Analysers*, February 2004. `http://www.imm.dtu.dk/cs_LySa/analysis1.html`; accessed January 20, 2005.

[22] G. Burns. *The Value-Passing Translator User Guide*. Bell Labs, Lucent Technologies, December 1996.

[23] R. Cavada, A. Cimatti, E. Olivetti, M. Pistore, and M. Roveri. *NuSMV 2.1 User Manual*. `http://nusmv.irst.itc.it`; accessed January 20, 2005.

[24] D. Chaum, A. Fiat, and M. Naor. Untraceable electronic cash. In *Proceedings of Advances in Cryptography – CRYPTO '88*, LNCS 403, pages 319–327, 1990.

[25] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV2: an opensource tool for symbolic model checking. In *Computer-Aided Verification: 14th International Conference*, LNCS 2404, pages 359–364, 2002.

[26] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model verifier. In *Computer-Aided Verification: 11th International Conference*, LNCS 1633, pages 495–499, 1999.

[27] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[28] E. Clarke and I. Draghicescu. Expressibility results for linear time and branching time logics. In *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency: School/Workshop*, LNCS 354, pages 428–437, 1989.

[29] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *TOPLAS*, 8(2):244–263, 1986.

[30] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 1999.

[31] R. Cleaveland, T. Li, and S. Sims. *The Concurrency Workbench of the New Century Version 1.2*. Department of Computer Science, Stony Brook University, July 2000.

[32] B. Cox, J.D. Tygar, and M. Sirbu. Netbill security and transaction protocol. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, pages 77–88, 1995.

[33] R. De Nicola and F. Vaandrager. Action versus state based logics for transition systems. In *Semantics of Systems of Concurrent Processes*, LNCS 469, pages 407–419, 1990.

[34] C. Fencott. *Formal Methods for Concurrency*. International Thomson Computer Press, 1996.

[35] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Addison-Wesley, 1997.

[36] U. Frendrup and J.N. Jensen. Checking for open bisimilarity in the π-calculus. Technical report, Department of Computer Science, Aalborg University, 2001. `http://www.cs.auc.dk/research/FS/ny/PR-pi`; accessed January 20, 2005.

[37] U. Frendrup and J.N. Jensen. *A User Manual for the OBC Workbench*. Department of Computer Science, Aalborg University, 2001. `http://www.cs.auc.dk/research/FS/ny/PR-pi`; accessed January 20, 2005.

[38] S. Gnesi, D. Latella, and M. Massink. Model checking UML statechart diagrams using JACK. In *4th IEEE International Symposium on High-Assurance Systems Engineering*, pages 46–55. IEEE Computer Society, 1999.

[39] S. Gnesi, D. Latella, and M. Massink. Modular semantics for a UML statechart diagrams kernel and its extension to multicharts and branching time model-checking. *The Journal of Logic and Algebraic Programming*, 51(1):43–75, 2002.

[40] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[41] Object Management Group. *OMG XML Metadata Interchange (XMI) Specification Version 1.2*, January 2002. `http://www.omg.org/technology/documents/formal/xmi.htm`; accessed January 20, 2005.

[42] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[43] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1998.

[44] D. Harel and A. Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[45] D. Harel, A. Pnueli, J. Schmidt, and R. Sherman. On the formal semantics of statecharts. In *Proceedings of 2nd IEEE Symposium on Logic in Computer Science*, pages 54–64. IEEE Computer Society, 1987.

[46] N. Heintze, J.D. Tygar, J. Wing, and H.C. Wong. Model checking electronic commerce protocols. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, pages 147–164, 1996.

[47] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[48] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[49] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, 2000.

[50] C.B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, second edition, 1990.

[51] V.S.W. Lam and J. Padget. Automated equivalence checking of UML statechart diagrams using the MWB. Submitted for publication.

[52] V.S.W. Lam and J. Padget. An integrated environment for communicating UML statechart diagrams. To appear in Proceedings of 3rd ACS/IEEE International Conference on Computer Systems and Applications.

[53] V.S.W. Lam and J. Padget. Formalization of UML statechart diagrams in the π-calculus. In *Proceedings of 2001 Australian Software Engineering Conference*, pages 213–223. IEEE Computer Society, 2001.

[54] V.S.W. Lam and J. Padget. Analyzing equivalences of UML statechart diagrams by structural congruence and open bisimulations. In *Proceedings of 2003 IEEE Symposium on Human Centric Computing Languages and Environments*, pages 137–144. IEEE Computer Society, 2003.

[55] V.S.W. Lam and J. Padget. On execution semantics of UML statechart diagrams using the π-calculus. In *Proceedings of the International Conference on Software Engineering Research and Practice*, pages 877–882. CSREA Press, 2003.

[56] V.S.W. Lam and J. Padget. Analyzing execution semantics of statecharts variants. In *Proceedings of 8th World Multi-Conference on Systemics,Cybernetics and Informatics*, volume 1, pages 474–478. IIIS, 2004.

[57] V.S.W. Lam and J. Padget. Formal specification and verification of the SET/A protocol with an integrated approach. In *Proceedings of 2004 IEEE International Conference on E-Commerce Technology*, pages 229–235. IEEE Computer Society, 2004.

[58] V.S.W. Lam and J. Padget. Symbolic model checking of UML statechart diagrams with an integrated approach. In *Proceedings of Eleventh IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 337–346. IEEE Computer Society, 2004.

[59] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[60] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11:637–664, 1999.

[61] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *3rd International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, pages 331–347. Kluwer, 1999.

[62] J. Lilius and I.P. Paltor. Formalising UML state machines for model checking. In *UML'99*, LNCS 1723, pages 430–445, 1999.

[63] G. Lowe and B. Roscoe. Using CSP to detect errors in the TMN protocol. *IEEE Transactions on Software Engineering*, 23(10):659–669, 1997.

[64] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 User Manual*, May 2003. `http://www.fsel.com/fdr2_download.html`; accessed January 20, 2005.

[65] S. Lu and S. Smolka. Model checking the secure electronic transaction (SET) protocol. In *7th International Symposium on MASCOTS*, pages 358–364. IEEE Computer Society, 1999.

[66] G. Lüttgen, M. von der Beeck, and R. Cleaveland. Statecharts via process algebra. In *10th International Conference on Concurrency Theory*, LNCS 1664, pages 399–414, 1999.

[67] G. Lüttgen, M. von der Beeck, and R. Cleaveland. A compositional approach to statechart semantics. In *ACM SIGSOFT Eight International Symposium on the Foundations of Software Engineering*, pages 120–129, 2000.

[68] A. Maggiolo-Schettini, A. Peron, and S. Tini. Equivalences of statecharts. In *CONCUR '96*, LNCS 1119, pages 687–702, 1996.

[69] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR '92*, LNCS 630, pages 550–564, 1992.

[70] MasterCard and VISA. *SET Secure Electronic Transaction Books 1–3*, May 1997.

[71] K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, 1992.

[72] E. Mikk, Y. Lakhnech, M. Siegel, and G. Holzmann. Implementing statecharts in Promela/SPIN. In *2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, pages 90–101. IEEE Computer Society, 1999.

[73] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[74] R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification, Proceedings of International NATO Summer School*, volume 94, pages 203–246. Springer-Verlag, 1993.

[75] R. Milner. Turing award lecture: Elements of interaction. *Communications of the ACM*, 36(1):78–89, 1993.

[76] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[77] R. Milner, J. Parrow, and D. Walker. A calculus of mobile process (Parts I and II). *Information and Computation*, 100:1–77, 1992.

[78] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[79] J.J. Odell, H.V.D. Parunak, and B. Bauer. Representing agent interaction protocols in UML. In *Proceedings of the First International Workshop AOSE 2000*, LNCS 1957, pages 121–140, 2001.

[80] D. O'Mahony, M. Peirce, and H. Tewari. *Electronic Payment Systems*. Artech House, 1997.

[81] D. O'Mahony, M. Peirce, and H. Tewari. *Electronic Payment Systems for e-Commerce*. Artech House, second edition, 2001.

[82] OMG. OMG Unified Modeling Language specification version 1.5, March 2003. `http://www.omg.org`; accessed January 20, 2005.

[83] OMG. UML 2.0 superstructure specification, August 2003. `http://www.omg.org`; accessed January 20, 2005.

[84] OSI. Open source initiative. `http://www.opensource.org`; accessed January 20, 2005.

[85] M.H. Park, K.S. Bang, J.Y. Choi, and I. Kang. Equivalence checking of two statechart specifications. In *11th International Workshop on Rapid System Prototyping*, pages 46–51. IEEE Computer Society, 2000.

[86] T. Parr. *An Introduction to ANTLR*. `http://www.cs.usfca.edu/~parrt/course/652/lectures/antlr.html`; accessed January 20, 2005.

[87] J. Parrow. An introduction to the π-calculus. In A. Bergstra, J.A. Ponse and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 8, pages 479–543. Elsevier Science, 2001.

[88] A. Pnueli and M. Shalev. What is a step: On the semantics of statecharts. In *TACS '91*, LNCS 526, pages 244–264, 1991.

[89] R. Pooley and P. Stevens. *Using UML: Software Engineering with Objects and Components*. Addison-Wesley, 1999.

[90] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall, 1991.

[91] R. Pucella. *Notes on Programming Standard ML of New Jersey (version 110.0.6)*. Department of Conputer Science, Cornell University, January 2001. `http://www.cs.cornell.edu/riccardo/smlnj.html`; accessed January 20, 2005.

[92] P. Quaglia. The π-calculus: Notes on labelled semantics. *Bulletin of the EATCS*, 68, June 1999.

[93] J. Rambaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[94] A. Ramirez, P. Vanpeperstraete, A. Rueckert, K. Odutola, and J. Bennett. *A tutorial and reference description of ArgoUML*, 2002. `http://argouml.tigris.org`; accessed January 20, 2005.

[95] W. Rankl and W. Effing. *Smart Card Handbook*. John Wiley & Sons, 1997.

[96] A. Ray, R. Cleaveland, and A. Skou. An algebraic theory of boundary crossing transitions. *Electronic Notes in Theoretical Computer Science*, 115:69–88, 2005.

[97] I. Ray and I. Ray. Failure analysis of an e-commerce protocol using model checking. In *Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, pages 176–183. IEEE Computer Society, 2000.

[98] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analyzing UML active classes and associated state machines – a lightweight formal approach. In *FASE 2000*, LNCS 1783, pages 127–146, 2000.

[99] R.L. Rivest and A. Shamir. Payword and Micromint: Two simple micropayment schemes. In *Proceedings of Security Protocols: International Workshop*, LNCS 1189, pages 69–87, 1997.

[100] A. Romão and M.M. da Silva. An agent-based secure internet payment system for mobile computing. In *Proceedings of TREC '98*, LNCS 1402, pages 80–93, 1998.

[101] D. Sangiorgi. *Expressing Mobility in Process Algebras: First Order and Higher-Order Paradigms*. PhD thesis, Computer Science Department, University of Edinburgh, 1993.

[102] D. Sangiorgi. A theory of bisimulation for the π-calculus. In *CONCUR '93*, LNCS 715, pages 127–142, 1993.

[103] D. Sangiorgi and D. Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[104] S.A. Seshia, R.K. Shyamasundar, A.K. Bhattacharjee, and S.D. Dhodapkar. A translation of statecharts to Esterel. In *FM'99*, LNCS 1709, pages 983–1007, 1999.

[105] O. Slotosch. Quest: Overview over the project. In *FM–Trends 98*, LNCS 1641, pages 346–350, 1999.

[106] J.M. Spivey. *The Z notation: A Reference Manual*. Prentice Hall, second edition, 1992.

[107] P. Stevens. *The Edinburgh Concurrency Workbench User Manual (Version 7.1)*. Laboratory for Foundations of Computer Science, University of Edinburgh, July 1999.

[108] Sun. Sun's Java 2 platform standard edition (J2SE) web site. `http://java.sun.com/j2se`; accessed January 20, 2005.

[109] D. Varró. A formal semantics of UML statecharts by model transition systems. In *ICGT 2002*, LNCS 2505, pages 378–392, 2002.

[110] B. Victor. *A Verification Tool for the Polyadic π-Calculus*. Department of Computer Systems, Uppsala University, 1994. Licentiate thesis.

[111] B. Victor and F. Moller. The mobility workbench: A tool for the π-calculus. In *CAV '94*, LNCS 818, pages 428–440, 1994.

[112] M. von der Beeck. A comparison of statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS 863, pages 128–148, 1994.

[113] M. von der Beeck. Formalization of UML-statecharts. In *UML 2001*, LNCS 2185, pages 406–421, 2001.